Om syntaks og syntaksanalyse
IN 211

Stein Krogdahl
Christian-Emil Ore

August 1993

Forord

Dette kompendiet er ment å skulle dekke omtrent ett vekttall av pensum til kurset IN 211, som i alt har tre vekttall. Den læreboka som brukes i IN 211 har svært lite om syntaks, og ut fra at vi ikke kan *forutsette* særlig mye teoretiske kunnskaper om grammatikker og språk, har det falt naturlig å lage et spesielt tilpasset kompendium. Vi har her lagt vekt på å gi en oversikt over forskjellige klasser av språk, og hvordan disse praktisk blir brukt i forbindelse med kompilatorer og definisjon av programmeringsspråk.

Den mer teoretiske bakgrunn for grammatikker, språk og de tilhørende "automater" (maskiner) blir gitt i kurset MA/IN 118. Vi skal i dette kompendiet referere noen setninger derfra, og forsøke å sette dem inn i en praktisk sammenheng. Selv om det kan være en fordel å ha tatt, eller ta i parallell, noe av denne teorien, er det ikke meningen at dette skal være en forutsetning.

Studenter som har fulgt de grunnleggende kursene ved IfI vil allerede ha vært borte i måter å beskrive syntaks, og måter å gjøre syntaksanalyse. I særdeleshet gjelder dette kurset IN 102, der syntaksen for språket Minila beskrives ved syntaksdiagrammer, og der syntaksanalysen gjøres ved den intuitivt besnærende "recursive descent" metoden. En del innledende armøvelser i dette kompendiet er derfor gjort meget korte, ut fra at studentene skulle ha bakgrunn nok til å kunne ta dette forholdsvis kjapt.

Det er viktig å være klar over at dette kompendiet ikke på noen måte gir en fullstendig oversikt over de teknikker som brukes ved syntaktisk analyse av programmer. For å kunne komme gjennom en så stor stoffmengde som mulig er det heller ikke lagt vekt på å føre bevisene i full detalj. I stedet har vi forsøkt å få fram den tankegangen og de hovedideene som ligger bak stoffet og resonnementene.

Til kurset IN 211 hører også et oppgavekompendium, der det blant annet er en del oppgaver og løsningsforslag til stoffet i dette kompendiet. Man skal i den forbindelse være klar over at oppgavekompendiet ble laget i forhold til en *tidligere* utgave av dette kompendiet, og en del sidehenvisniger osv. til dette kompendiet kan derfor være misvisende.

Som for de fleste kompendier, gjelder det også for dette at det ble til under hardt tidspress. Det vil derfor nødvendigvis være en rekke store og små trykkfeil, men forhåpenltigvis ikke så mange som kan skape diretke misforstuelser. Den pedagogiske tilretteleggingen vil nødvendigvis også kunne forbedres, og forfatterne og kursledelsen tar gjerne i mot kommentarer til framstillingen.

Innhold

Kapittel 1

Innledning

Dette kompendiet er ment som en utdyping og utvidelse av det lille som står om syntaks og språkdefinisjon i lærboka til IN 211 (Ghezzi og Jazayeri: *Programming Language Concepts*). Som bakgrunn bør man imidlertid også lese gjennom det som står der om dette tema, i begynnelsen av kapittel 3. Kapittel 1 og 2 i dette kompendiet er ment å dekke det samme stoffområdet på en noe utdypende måte. Vær oppmerksom på at vi i dette kompendiet av og til bruker en litt annen terminologi, men dette burde gjelde så få begreper og være så tydelig markert at det forhåpentligvis ikke skaper problemer. En tilsvarende advarsel gjelder forøvrig i forhold til den boka som brukes i MA/IN 118.

Når man skal beskrive et programmeringsspråk, deler man gjerne reglene for hva som er et lovlig program opp i to deler: På den ene side de regler som beskriver hva slags "form" et lovlig program skal ha, og på den annen side de mer fiklete reglene som vanligvis grovt sett sier at alle navn må være deklarart som ett eller annet, og at hver gang et navn brukes så må det brukes i overenstemmelse med det det er deklarart som. På slutten av kapittel 2 skal vi se at denne todelingen av reglene har en viss teoretisk begrunnelse.

Selv om grensen for hva som er regler for "form" og hva som er tilleggsregler av og til kan være litt flytende, vil denne oppdeligen stå rimelig klart fram i de fleste språkdefinisjoner. I tillegg vil selvfølgelig en slik definisjon også beskrive hva de forskjellige setningene i språket skal *bety*, og ofte er det naturlig å komme med tilleggskravene i forbindelse med dette. Man bør være klar over at de begreper som brukes for å betegne de forskjellige delene av en språkdefinisjon varierer ganske mye fra den ene kultur til den andre, og under gir vi derfor en liten utlegning om dette.

Dette kompendiet dreier seg først og fremst om de aspektene som har med den rene formen på programmene å gjøre, og dette vil alle være enig om at hører inn under begrepet "syntaks". I kapittel 2 vil vi se på noen forskjellige formalismer som brukes til å beskrive slik form.

I kapittel 3 vil vi se på teoretiske og praktiske sider av en noe forenklet formalisme for å beskrive slik form, nemlig de regulære grammatikker og de tilsvarende regulære språk. Denne formalismen er velegnet både til å beskrive og til å lage algoritmer i forbindelse med de minste meningsbærende enheter i et program, så som navn, skilletegn og operatorer av forskjellige typer.

I kapittel 4 ser vi så på algoritmer for å kunne forstå strukturen i et program når man leser det symbol for symbol. Å "forstå strukturen" vil her grovt sett si at algoritmen klarer å gjenskape oppbygningen av den aktuelle setningen ut fra den formalismen som er brukt for å definere de lovlige former. Det å finne fram til strukturen på en gitt setning kalles gjerne *parsering* eller *syntaksanalyse*, og de tilsvarende algoritmer kalles *parseringsalgoritmer*.

I denne forbindelse skal vi også klassifisere språk og grammatikker ut fra hvor krevende det er å forstå dem. For disse klassene skal vi så skissere hvordan man kan avgjøre om en språkbeskrivelse er med i den aktuelle klassen, og hvordan man i så fall kan utforme en parseringsalgoritme for dette språket.

1.1 Bruk av begrepene syntaks og semantikk

Folk som arbeider med språk i forskjellige former bruker gjerne begepene *syntaks* og *semantikk* for å skille mellom forskjellige sider av språket og teoriene rundt det. Selv om det er enighet om at syntaks stort sett skal dekke det som har med "formen" av setningene i språket å gjøre, mens semantikk skal dekke det som har å gjøre med "meningen" med setningene, så er det rike muligheter til å bruke begrepene så pass forskjellige at det kan virke forvirrende.

Syntaks og semantikk er opprinnelig språkvitenskaplige termer, men folk som arbeider med naturlige språk bruker dem ofte noe anderledes enn databehandlerne. I det følgende vil vi først se hvordan syntaks og semantikk brukes av databehandlere. Deretter vil vi kort skissere den mer tradisjonelle bruken innen filologien.

De språkene databehandlere arbeider med er kunstige språk, laget for å beskrive ting innen et avgrenset saksområde. De språkene vi skal tenke på her er de som skal beskrive en utførelse på en datamaskin, altså de tradisjonelle programmeringsspråkene. Når man snakker om syntaks i disse språkene mener man gjerne den rene form, altså f.eks. at en if-setning starter med ordet 'if' at det deretter kommer noe som har form av en *betingelse* osv. Det man da *ikke* inkluderer i syntaks er alt som har med *at* ting er deklarert, og at det er deklarert på en måte som stemmer overens med den måten det brukes på.

Dette at alle navn skal være deklarert, og at bruken skal stemme med deklarasjonen, sier databehandlerne at har noe å gjøre med hva navnene "betyr". Det er derfor rimelig å rubrisere dette til semantikk selv om det er klart at slike ting må sjekkes før en kan snakke om et riktig eller meningsfylt program. Denne delen av semantikken kalles ofte den *statiske* semantikken til et programmeringsspråk fordi den beskriver hva som kan sjekkes før utførelsen starter.

Som motsetning til dette kalles reglene om hva som skal skje under utførelsen for et gitt (riktig) program for den *dynamiske* semantikken.

Folk som driver med naturlige språk har tradisjonelt et romsligere syntaksbegrep. Syntaks rommer alt som har med måten et språks ord og setninger kan ordnes og forbindes på. De "konteksavhengige" reglene, f.eks. for bruk av pronomen, ansees som syntaktiske regler og ikke semantiske som i databehandlingen. Det må i den forbindelse sies at det er lite meningsfylt å snakke om utførelsen av en setning i et naturlig språk.

Semantikk omhandler ord og setningers meningsinnhold og kan for naturlige språk fort få en filosofisk karakter. Naturlige språk er ikke formelt definerte og avgrensede slik som formelle, kunstige språk. Grensen mellom begrepene syntaks og semantikk i filologien er relativt uklar.

Vi skal endelig se litt på språkbruken til en mer spesialisert gruppe databehandlere, nemlig de som mer eksplisitt arbeider med kompilatorer og definisjon av programmeringsspråk. Disse deler gjerne de syntaktiske reglene videre opp i to grupper, nemlig de "leksikalske reglene" og "den egentlige syntaks". Hva dette innbærer skal vi komme tilbake til i senere kapitler, men bare her si at de leksikalske reglene er regler for å bygge opp de minste meningsbærende enheter i programmet, så som navn, tall, nøkkelord, aritmetiske og logiske operatorer osv. Reglene for å bygge opp disse er ofte ganske enkle, og de kan som regel beskrives av en spesielt enkel type grammatikker, kalt *regulære grammatikker*. Det man da kaller de syntaktiske reglene, er hvordan disse kan settes sammen til riktige programmer. Den egentlige syntaksen er reglene for hvordan program kan bygges opp fra de leksikalske enhetene.

I dokumenter som definerer programmeringsspråk finner man gjerne igjen dette skillet ved at det er et eget kapittel som beskriver reglene for å danne de leksikalske enhetene, og at de øvrige kapitlene bare beskiver hvordan fullstendige programmer kan bygges opp ved hjelp av slike enheter. I kompilatorer finner vi også skillet igjen, ved at det gjerne er en egen "preprosessor" som leser programmet og deler det opp i en sekvens av leksikalske enheter, som så leveres på vakker form til selve kompilatoren (slik som i Minilakompilatoren i IN 102, for de som har det kurset).

Kapittel 2

Språk, og formelle beskrivelser av språk

Vi skal her se på forskjellige måter man kan beskrive syntaksen for et språk. Vi skal spesielt ha programmeringsspråk i tankene, men formalismene vi skal se på kan selvfølgelig like gjerne brukes til å beskrive dataformater og andre språkliknende strukturer. Hovedvekten er lagt på hvordan de såkalte *kontekstfrie språk* kan beskrives. En viktig subklasse av de kontekstfrie språk er de *regulære språk*. Disse nevnes bare i dette kapittelet, mens vi i neste kapittel skal se litt nærmere på dem.

Klassene av de kontekstfrie og de regulære språkene utgjør to av de i alt fire klassene man gjerne opererer med i formell språkteori. I slutten av dette kapittelet gir vi en kort oversikt over diise fire klassene og det hierarkiet de danner.

For å kunne snakke om et språk, må vi ha et *alfabet* som angir de tegnene eller symbolene som språkets setninger er bygget opp av. Når man holder på å beskrive de leksikalske enheter, så vil dette alfabetet gjerne bestå av de enkelte tegn som kan skrives på et tatstatur, mens når man snakker om de vanlige syntaktiske reglene i et programmeringsspråk, så tenker man seg gjerne at alfabetet består av de forskjellige leksikalske enhetene i språket. Vi skal generelt omtale elementene i det alfabetet som *symboler*. Sekvenser av symboler fra alfabetet skal vi kalle *strenger* over dette alfabetet.

Et *språk over et gitt alfabet* er generelt et utplukk av alle mulige strenger over alfabetet. De strengene som er plukket ut skal vi kalle språkets *setninger*. I de språk vi skal se på vil alle setninger ha endelig lengde, men det kan gjerne være uendelig mange setninger. Slike utplukk av strenger kan selvfølgelig beskrives på svært mange måter, men vi skal i det følgende se på hvordan man kan beskrive et språk ved såkalte BNF-grammatikker, samt på noen varianter av dette.

2.1 Rene BNF-grammatikker

Den mest kjente måten å beskrive et språk på er å bruke en såkalt BNF-grammatikk, der BNF står for "Backus-Naur-Form". Denne formalismen fordrer at vi innfører noen flere symboler, som grovt sett skal representere de mer eller mindre kompliserte deler som setningene i språket blir bygget opp av. For å skille disse symbolene fra de i det opprinnelige alfabetet, skal vi kalle dem for *metasymboler*, mens de opprinnelige symbolene kalles *grunnsymboler* (disse kalles også "terminal-symboler", og da kalles metasymbolene gjerne "ikketerminal-symboler"). Vi skal i eksempler skrive metasymboler med store bokstaver og grunnsymboler med små bokstaver eller med andre tegn. I læreboka brukes spissparenteser omkring metasymbolene, men det avstår vi fra her. (I stedet skal vi bruke spissparentesene til noe annet).

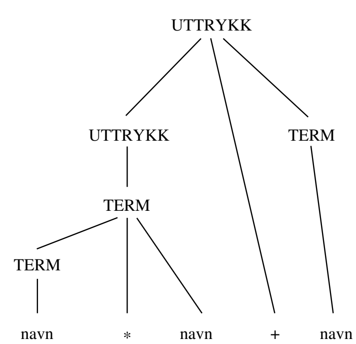
Som et eksempel på en BNF-grammatikk kan vi se på følgende fire linjer:

$$\begin{array}{lcl} \text{UTTRYKK} & \rightarrow & \text{UTTRYKK} + \text{TERM} \\ \text{UTTRYKK} & \rightarrow & \text{TERM} \\ \text{TERM} & \rightarrow & \text{TERM} * \textbf{navn} \\ \text{TERM} & \rightarrow & \textbf{navn} \end{array}$$

Hver linje kalles her en *produksjon* i grammatikken. Til venstre for pilen skal det alltid stå ett metasymbol (og ingen grunnsymboler). Pilen kan passelig leses som: "kan ha formen", og til høyre for denne skal det være en sekvens av grunn- og metasymboler som angir en lovlig form for metasymbolet på venstre side. Ideen er at man for hver av metasymbolene i slike former skal kunne sette inn en av de lovlige former for dette metasymbolet. Vi snakker gjerne om *venstresiden* og *høyresiden* av en produksjon.

Ett av metasymbolene må utpekes som *startsymbolet*, og dette representerer de former som selve setningene i språket kan ha. I grammatikken over er UTTRYKK startsymbolet, men i forbindelse med programmeringsspråk vil dette ofte ha navnet PROGRAM.

Setningene i språket fremkommer altså ved å starte med startsymbolet, og så for hvert metasymbol i de mellomformene man har, sette inn en av de alternative former dette kan ha. Setningene i språket er da definert som de former man kan komme til på denne måten, som bare består av grunnsymboler. Vi kaller en slik substitusjons-prosess som dette for en *avledning* fra startsymbolet til en ferdig setning, og en slik avledning kan mest naturlig representeres som et tre. Et slikt tre kalles gjerne et *syntakstre*, men kunne like gjerne vært kalt kalt et *avledningstre*. F.eks. vil setningen "**navn** $*$ **navn** $+$ **navn**" i grammatikken over kunne avledes ved følgende syntakstre:



I stedet for å angi en avledning i form av et syntakstre, kan man angi den som en sekvens av halvutviklede *setningsformer* som utvilker seg fra bare startsymbolet til en setning med bare grunnsymboler, og der vi i hvert steg bare anvender én produksjon. En slik avledning skrives ofte med dobbelt-pil i stedet for enkelt-pil mellom leddene, og en avledning av setningen over kan da f.eks. skrives slik:

$$\text{UTTRYKK} \Rightarrow \text{UTTRYKK} + \text{TERM} \Rightarrow \text{TERM} + \text{TERM} \Rightarrow$$
$$\text{TERM} * \textbf{navn} + \text{TERM} \Rightarrow \textbf{navn} * \textbf{navn} + \text{TERM} \Rightarrow \textbf{navn} * \textbf{navn} + \textbf{navn}$$

Ved å variere i hvilken rekkefølge vi utvikler de forskjellige metasymbolene kan vi selvfølgelig få mange forskjellige slike avledninger som tilsvarer det samme treet. For å få entydighet også her snakker man derfor om *venstreavledninger* der man hele tiden utvikler videre det metasymbolet som står lengst til venstre, og tilsvarende om *høyreavledninger*. Avledningen over er en venstreavledning.

Man tillater vanligvis at man har produksjoner med tom høyreside. Vi skal angi en slik høyreside med tegnet 'ϵ'.

Legg merke til at man gjerne kan la metasymbolet på venstresiden i en produksjon gå igjen én eller flere ganger i høyresiden. Vi kaller dette *rekursive produksjoner*, og om vi finner igjen metasymbolet helt til venstre i høyresiden snakker vi om *venstrerekursjon*, og tilsvarende snakker vi om *høyrerekursjon*. Grammatikken over har altså bare venstrerekursjon.

Endelig slår man ofte sammen de produksjonene som har samme venstreside, og angir de alternative høyresidene men en loddrett strek mellom. Grammatikken over kan da skrives slik:

$$\begin{array}{lcl} \text{UTTRYKK} & \rightarrow & \text{UTTRYKK} + \text{TERM} \mid \text{TERM} \\ \text{TERM} & \rightarrow & \text{TERM} * \textbf{navn} \mid \textbf{navn} \end{array}$$

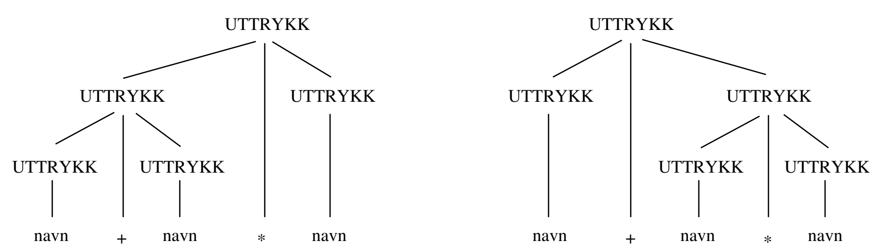
Merk at vi fremdeles skal si at denne grammatikken har *fire* produksjoner (og ikke bare to). Når grammatikken skrives på denne måten brukes ofte tegnet '::=' i stedet for '$\rightarrow$'. Vi skal imidlertid holde oss til pila også her.

2.1.1 Entydige og flertydige grammatikker

Dersom hver setning i språket kan avledes ved ett og bare ett syntakstre, sies grammatikken å være *entydig*, og i motsatt fall *flertydig*. For et språk kan det som regel gis både entydige og flertydige grammatikker. Den grammatikken vi har gitt over er entydig, men en flertydig grammatikk for det samme språket kan f.eks. være følgende:

$$\text{UTTRYKK} \rightarrow \textbf{navn} \mid \text{UTTRYKK} + \text{UTTRYKK} \mid \text{UTTRYKK} * \text{UTTRYKK}$$

Det er her lett å se at man kan gi to forskjellige syntaksstrær for visse setninger, f.eks. for den setningen vi så på tidligere:



Vi ser her at den opprinnelige grammatikken tvang treet til å trekke sammen '$*$' før '$+$', mens vi i den flertydige grammatikken kunne velge dette fritt. I forbindelse med programmeringsspråk forsøker man som regel å lage entydige grammatikker, fordi syntakstreet for programmet også gjerne vil være med å avgjøre hvordan selve programmet skal forstås. F.eks. vil jo de to trærne for setningen over naturlig føre til forskjellig beregning av uttrykket, som generelt vil gi forskjellig resultat.

Imidlertid er det slik at flertydige grammatikker for det samme språket ofte får både færre produksjoner og færre metasymboler, og de kan derfor bli mer oversiktlige. Dette framgår jo med all tydelighet av eksempelet over. Av og til beskriver man derfor språk ved hjelp av flertydige grammatikker, men gir tilleggsregler for hvilken tolkning som skal brukes der dette er nødvendig. En type slike tilleggsregler er de såkalte "presedensregler", og vi skal i slutten av kompendiet se hvordan dette kan benyttes.

2.2 Utvidet BNF-notasjon og syntaksdiagrammer

Det å beskrive et programmeringsspråk ved ren BNF-notasjon kan kreve nokså mange produksjoner, men er som regel slett ikke noen uoverkommelig oppgave. Vi så nettopp at én måte å forenkle en slik beskrivelse på kunne være å bruke flertydige grammatikker, men dette er som nevnt ofte ikke ønskelig. Vi skal her se på et par andre måter å beskrive syntaks på som vanligvis fører til mer kompakte og oversiktlige beskrivelser. Det helt kurant å omarbeide hver av disse formene til ren BNF-form, men det overlates til den interesserte leser å overbevise seg om dette.

2.2.1 Utvidet BNF

Utvidet BNF er en beskrivelsesmåte der vi tilbyr litt flere muligheter når man skal angi formene på høyresiden i produksjonene, og derved kan vi få sagt mer på en gang. Akkurat hvilke måter man tillater, og ikke minst hvordan de noteres, kan variere en del, men vi skal her se på noen typiske måter. Vi trenger da en type "metaparenteser" som vil være en del av BNF-syntaksen (liksom pilen og den loddrette streken), og til dette skal vi bruke spissparenteser. En avsluttende spissparentes kan også følges av tegnene '?', '$+$' eller '$*$', og disse er da også en del av språkbeskrivelsen. Man kan da f.eks. tillate følgende former hvor som helst i en høyreside:

$\{\ldots \mid \ldots \mid \ldots\}$	Man skal velge ett av alternativene. Vi utvider altså bruken av den loddrette streken.
$\{\ldots\}^?$	Det inni parentesen kan taes med eller kuttes ut. Dette angis ofte også med hakeparenteser.
$\{\ldots\}^+$	Det inne i parentesen skal gjentas en eller flere ganger.
$\{\ldots\}^*$	Det inne i parentesen skal gjentas null eller flere ganger.

Om man nå skal angi syntaksen til en passelig if-setning (ikke Simulas, men en med setningssekvenser inne i 'if – endif') og en passelig variabel-deklarasjon, kan dette nå f.eks. gjøres slik:

$$\begin{array}{lll} \text{IF-SETN} & \rightarrow & \textbf{if}\ \text{BETINGELSE}\ \textbf{then}\ \{\text{SETN ; }\}^*\ \{\textbf{else}\ \{\ \text{SETN ; }\}^*\}^?\ \textbf{endif} \\ \text{VAR-DEKL} & \rightarrow & \{\ \textbf{integer}\ |\ \textbf{real}\ |\ \textbf{boolean}\}\ \textbf{navn}\ \{\ ,\ \textbf{navn}\}^* \end{array}$$

Mer avanserte former for slik meta-syntaks finnes også, der man f.eks. kan angi at en viss konstruksjon ikke bare skal gjentas, men gjentas med et visst grunnsymbol i mellomrommene. Da må både grunnsymbolet og hvilken gjentakelsestype man ønsker, angis.

Begrepet entydighet lar seg nokså direkte overføre til denne typen grammatikk. I tillegg til at det bare må være ett tre (i en passelig utvidet forstand) for hver setning, må vi da også forlange at det bare er én måte å gjøre hvert av de aktuelle valg og repetisjoner på innen hver høyreside. Følgende produksjon fører f.eks. til at grammatikken ikke er entydig i denne betydning:

$$\text{SETN-SEKV} \rightarrow \{\text{SETN}\ |\ \text{SETN}\ \{;\ \text{SETN}\}^*\}$$

2.2.2 Syntaksdiagrammer

En annen populær beskrivelsesform er å bruke såkalte *syntaksdiagrammer*, og ideen her er at vi tegner hver høyreside ut som en slags rettet graf, med ett start-punkt og ett slutt-punkt. På kantene i grafen, gjerne inne i spesielle bokser, tegnes det så inn grunnsymboler og metasymboler, og enhver sekvens av symboler man kan danne ved å spasere gjennom grafen *med* pilene fra start- til sluttpunktet er en lovlig høyreside for den aktuelle produksjonen.

Denne beskrivelsesmåten blir ofte meget oversiktlig, og man kan som regel stappe svært mye inn i én "høyreside" uten at den blir vanskelig å lese. Syntaksen for Pascal er helt fra begynnelsen blitt beskrevet på denne måten. For å skille mellom metasymboler og grunnsymboler brukes gjerne firkantede og avrundede bokser, men hva som angir hva varierer. Som et eksempel gjengir vi under syntaksen for setninger i språket Minila (slik det var høsten 1991 i IN-102).

Begrepet entydighet kan også lett utvides til syntaksskjemaer. En språkbeskrivelse er entydig dersom hver setning bare kan produseres ved å sette sammen og gå gjennom syntaksskjemaene på en og bare én måte. Dette innebærer bl.a. at hvert skjema bare kan produsere "sin del" på én måte.

2.3 Klassifisering av språk

BNF-formalismen og dens avarter er bare én måte å beskrive språk på, og det viser seg at den også har sine klare begrensninger. Om vi bryter ut av denne formalismen kan vi lett angi språk som ikke kan beskrives ved BNF-formalsimen. Et enkelt slikt eksempel er språket med grunnsymbolene a, b og c, der en setning er med i språket dersom den først har et antall a-er som så er fulgt av like mange b-er, og avsluttet med like mange c-er. Dette språket angis gjerne ved formlen $a^n b^n c^n$. Beviset for at dette språket ikke kan beskrives ved en BNF-grammatikk skal vi ikke gå inn på her.

Den klassen av språk vi kan beskrive ved BNF-notasjon, eller ved en passelig avart av denne, kalles *kontekstfrie språk*. I teorien for formelle språk (som gjennomgås nærmere in

MA/IN118) opereres det gjerne i alt med fire klasser av språk, der hver klasse er en ekte restriksjon i forhold til den neste. De kontekstfrie språk kommer et stykke ned på stigen.

Nedenfor er det gitt en kort karakteristikk av de fire språkklassene. Merk altså at vi beskriver disse ved å si at språkene i dem genereres av BNF-aktige grammatikker, der vi tillater større frihet på *venstresiden* i BNF-produksjonene. I motsetning til hva vi opplevde med våre tidligere variasjoner av høyresidene, vil altså større frihet på venstresiden gi en ekte utvidelse med hensyn til hva slags språk man kan beskrive.

Type 0

Den mest generelle klassen består av de språk som kan beskrives av en BNF-aktig grammatikk der også venstresidene tillates å være et vilkårlig antall grunn og metasymboler, og der reglen er at enhver forekomst av en venstreside i en mellomform kan erstattes av den tilsvarende høyreside. Denne typen språk kalles bare *type 0*.

Type 1

Den neste klassen består av språk som kan beskrives av grammatikker der det i tillegg kreves at høyresideni hver produksjon er minst like lang som venstresiden (slik at den mellomformen man har aldri blir kortere under en avledning). Språk som kan beskrives av slike grammatikker kalles også *kontekstsensitive*. Vær imidlertid oppmerksom på at i en del bøker brukes termen kontekstsensitivt mer generelt om språk som ikke er kontekstfrie.

Type 2

Den tredje klassen består av de språk som kan beskrives av vanlige BNF-grammatikker, dvs. av grammatikker der venstresiden består kun av ett metasymbol. Språkene i denne klassen er altså de *kontekstfrie*.

Type 3

Den siste og minste klassen består av de språk som kan beskrives med BNF-produksjoner der hver høyreside enten bare har grunnsymboler, eller i tillegg har ett metasymbol som står helt bakerst. (Helt forrest virker like bra bare man er konsekvent). Slike språk kalles *regulære*.

I teorien for formelle språk (som gjennomgås i MA/IN 118) viser man at disse klassene av språk kan karakteriseres også på andre måter. De regulære språk kan f.eks. defineres som de språk som kan "kjennes igjen" av en datamaskin med et endelig lager (en endelig automat). Med "kjennes igjen" mener vi her at vi kan lage en slik endelig automat som innen endelig tid kan svare på om en gitt setning er med i språketeller ikke. Dette må selvfølgelig for teoretiske formål presiseres ytterligere, men det skal vi ikke gå inn på her.

Det å gjenkjenne kontekstfrie språk krever en mer komplisert maskin. Her kan man ikke lenger klare seg med en datamaskin med et endelig lager, men må i tillegg ha et stakklager, der stakken må kunne så dyp man vil. En slik maskin kalles ofte en "push-down-automat". Her er det også mange tekniske detaljer (bl.a. med hensyn til "deterministisk" eller "ikkedeterministisk") som behandles utførlig i MA/IN 118.

Grovt sett kan man si at de kontekstfrie språk er de som har en eller annen "parentetisk struktur", altså at konstruksjoner man har begynt på inne i en annen konstruksjon, også må avsluttes før den ytre konstruksjonen avsluttes. I programmeringsspråk er nettopp blokkstruktur og uttrykk med parenteser eksempler på slike strukturer. Det er vel forøvrig intuitivt rimelig at den datastrukturen som skal til for å håndtere en parentetisk struktur, nettopp er en stakk.

Som nevnt er det ikke vanskelig å angi språk som *ikke* er kontekstfrie, og vi anga et enkelt slikt over. Mer interessant i vår sammenheng er det at om vi forsøker å uttrykke vanlige deklarasjonsregler i programmeringsspråk (om at alt må være deklarert osv.) ved hjelp av BNF-produksjoner, så viser dette seg å være umulig. Har språket slike regler (og det har jo de fleste programmeringsspråk) er det altså ikke lenger kontekstfritt.

Heri ligger også begrunnelsen for at man gjerne to-deler beskrivelsen av programmeringsspråk. BNF-grammatikker er så enkle og intuitive å ha med å gjøre at det er svært behagelig å beskrive så mye som mulig (de "kontekstfrie deler") av språket ved slike teknikker. Det man altså rent teknisk gjør er å beskrive med BNF-notasjon et språk som inneholder mange flere setninger enn de som faktisk er riktige programmer, nemlig alle som hvertfall har riktig "struktur".

Så må man da bruke et annet uttrykksmiddel (gjerne engelsk eller norsk) til å stille ytterligere krav til et riktig program, for dermed å sile ut den endelige mengde av riktige programmer. Man skal her legge merke til at i tillegg til å beskrive hva som er programmer med riktig struktur, så vil også BNF-bekrivelsen gi et *begrepsapparat og en terminologi* som gjør det lett å angi de ytterligere krav som skal til. I uttrykks-språket over kan vi f.eks. ut fra syntaksen snakke om "termer" og "navn", og tilsvarende vil vi i vanlige programmeringsspråk kunne snakke om "deklarasjoner", "if-setninger", "ytterste blokken" etc. etc. BNF-syntaksen skaper altså den grovstrukstur som gjør det mulig å uttrykke de mer kompliserte krav som stilles til et riktig program.

Man kan selvfølgelig spørre seg om ikke det fullstendige regelsettet for vanlige programmeringsspråk heller kunne beskrives ved type 1 eller type 0 grammatikker, og at man dermed kunne bruke slike mer generelle grammatikker for å beskrive språkene fullt ut. Svaret er at type 1 grammatikker i de aller fleste tilfeller rent formelt ville være mer enn godt nok, men at de språkbeskrivelsene vi da ville få ville bli svært kryptiske og voluminøse i forhold til den renheten vi oppnår med kontekstfrie språk.

Et annet argument mot dette er at det har vist seg greiest å la kompilatorer behandle de kontekstfrie og de "kontekstsensitive sider" av språket på nokså forskjellige måter. Det å lage den delen av en kompilator som tar seg av de kontekstfrie sider kan lages mer eller mindre automatisk (noe vi skal se mer på i fortsettelsen) ut fra en BNF-beskrivelse. Resten av kompilatoren blir imidlertid oftest programmert ut fra mer intuitive betrakninger.

Når vi senere i kompendiet skal se på teknikker for å lage den delen av en kompilator som analyserer de kontekstfrie deler av språket, kan vi merke oss at den lagerstrukturen som behøves er en stakk, noe som nettopp stemmer med det teoretiske resultat med en "push-down-automat" som vi omtalte over. For å kunne kjenne igjen kontekstsensitive (type 1) språk trenger man imidlertid noe kraftigere enn en slik stakkmaskin. Maskinen må da ha et lager som kan aksesseres i den rekkefølge man ønsker, og som også kan utvides etter behov. En svært enkel variant av en slik maskin er de såkalte Turing-maskiner, der lageret består av en uendelig lang array med et lese- og skrivehode som kan flyttes fram og tilbake én posisjon av gangen. Ellers er jo våre vanlige datamaskiner, i den grad deres lager kan utvides etter behov, også en variant av slike maskiner, som også har den fordelen at de er vesentlig mer behagelige å programmere.

Type 0 språk er av mer teoretisk interesse. Vi skal bare nevne at det er en prinsipiell forskjell mellom mekanismer for å kjenne igjen kontekstsensitive språk og type 0 språk. For et kontekstsensitivt språk vil det alltid finnes et program som etter å ha lest input vil stoppe og si om den leste sekvensen er en setning i språket eller ikke. For et type 0 språk vil det alltid finnes et program som stopper når input er en setning i språket. Men det er ikke nødvendigvis slik at det finnes et program som stopper om input ikke er en setning i språket.

Kapittel 3

Regulære språk

De regulære språk danner, som nevnt i forrige kapittel, en (ekte) underklasse av av de kontekstfrie språk. Regulære språk og teorien rundt dem er viktige innen databehandling. F.eks. brukes såkalte "regulære uttrykk" ofte til å spesifisere det mønster (med visse variasjoner) man vil lete etter i en tekst. I vår sammenheng er det imildertid like interessant at de minste meningsbærende enhetene i programmeringsspråk vanligvis kan beskrives som (setninger i) regulære språk. Som vi skal se siden kommer teorien omkring regulære språk også inn i metoder for å kjenne igjen kontekstfrie språk.

Vi konstaterte i forrige kapittel at de regulære språkene er de språk som kan kjennes igjen av en maskin med et endelig avgrenset lager. En spesiell interessant variant av disse maskinene kalles de *endelige automater*. Det finnes igjen to typer av disse, nemlig de *deterministiske* og de *ikkedeterministiske*, og begge disse skal vi se mer på om et øyeblikk.

Det finnes en rekke måter å beskrive de regulære språk på, som man uten større problemer kan vise er ekvivalente. Vi gjengir her først en liste av slike måter, og går så siden litt inn på noen av dem:

(A) De språk som kan beskrives med BNF-produksjoner der hver høyreside enten bare har grunnsymboler, eller har ett metasymbol som står helt bakerst. (Helt forrest virker like bra bare man er konsekvent, og man kan også legge på en restriksjon om at hver høyreside ikke kan inneholde mer enn ett grunnsymbol).

(B) De språk som kan beskrives ved en eneste produksjon i *utvidet* BNF, der høyresiden bare inneholder grunnsymboler. Slike høyresider kalles gjerne *regulære uttrykk*.

(C) De språk som kan beskrives ved et eneste syntaksdiagram, der det i diagrammet bare angis grunnsymboler.

(D) De språk som kan kjennes igjen av endelige ikkedeterministiske automater.

(E) De språk som kan kjennes igjen av endelige deterministiske automater.

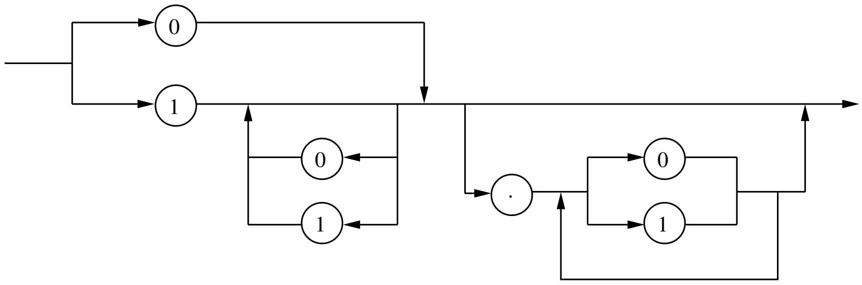
Som et eksempel på regulære språk skal vi bruke en typisk syntaks for tall som kan ha desimalpunktum, og som må ha minst ett siffer foran punktum, og også ett bak dersom det er et punktum. For enkelhets skyld skal vi anta at vi skal beskrive binære tall der sifrene er '0' og '1' (ordet "desimalpunktum" er jo da litt misvisende), men vi skal forlange at om det er flere enn ett siffer foran punktum, så skal det første være '1' (altså ikke innledende nuller). Under har vi beskrevet dette språket på de forskjellige måtene angitt over.

(A) TALL $\rightarrow$ 0 FP | 1 IFP
IFP $\rightarrow$ 1 IFP | 0 | FP
FP $\rightarrow$ ϵ | . EP
EP $\rightarrow$ 0 | 1 | 0 EP | 1 EP

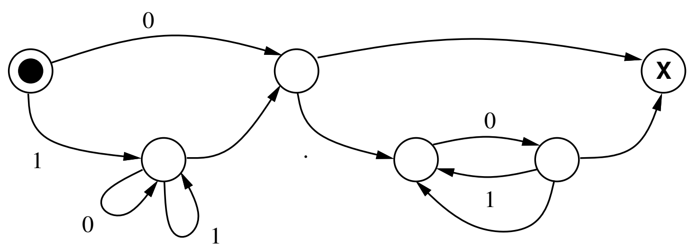
De navnene vi har gitt symbolene skulle indikere følgende: IFP: "inni foran punktum", FP: "foran punktum", EP: "etter punktum".

(B) TALL $\rightarrow$ $\{0 \mid 1 \{0 \mid 1\}^{*}\} \{ . \{0 \mid 1\}^{+}\}^{?}$

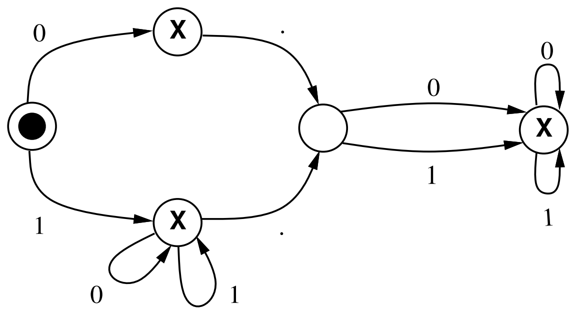
(C)



(D)



(E)



Alt unntatt (D) og (E) skulle være ting man kjenner fra før. Leseren oppfordres til å kontrollere at det er det riktige språket som beskrives av de tre første formene. Vi skal i det følgende se litt på de to siste formene.

3.1 Ikkedeterministiske automater

En ikkedeterministisk automat er egentlig et begrep som er svært nært et syntaksdiagram med bare grunnsymboler, bare at vi har skiftet litt om på "kanter og noder". For begge vil de lovlige setningene være de som kan produseres ved å følge pilretningen fra startpunktet (eller startnoden) til et lovlig sluttpunkt, og lage en streng av de symboler man passerer.

Konverteringen fra et syntaksdiagram til en ikkedeterministisk automat kan rett og slett gjøres ved å legge inn noder i hvert forgreningspunkt, ved starten og ved slutten, og ellers "mellom" hvert grunnsymbol i diagrammet. Vi lar så diagrammets linjer bli kanter mellom nodene, og noen av disse kantene vil da være merket med et grunnsymbol, mens andre vil være uten symbol. Automaten under (D) er, med et par forenklinger, produsert fra (C) på denne måten.

En generell ikkedeterministisk automat kan beskrives slik: En rettet graf der noen av kantene er merket med et grunnsymbol mens andre er umerkede. Nodene i denne grafen kalles gjerne automatens *tilstander*, mens kantene uten merke kalles ϵ-kanter (ϵ står gjerne for den tomme setning). Merk at det ikke er noe krav om at to kanter som går ut fra samme tilstand er forskjellig merket.

Én av tilstandene skal være valgt som *starttilstanden*, og ved overgang fra syntaksdiagrammet er dette den som som tilsvarer startpunktet i diagrammet. Videre skal et antall tilstander være valgt som *sluttilstander*, som også ofte kalles *aksepterende* tilstander. Ved overgang fra syntaksdiagrammet velges den tilstand som tilsvarer sluttpunktet i diagrammet som eneste sluttilstand.

I eksemplene (D) og (E) over er starttilstanden merket med en prikk og sluttilstandene med et kryss.

Dersom vi vil generere forskjellige lovlige setninger er både syntaksdiagrammer og ikkedeterministiske automater greie å bruke. De har også den fordel at de er greie å sette opp, da man har full frihet til å putte på nye veier gjennom grafen, uten å tenke på de som er der fra før.

Når vi snakker om *automater* har vi imidlertid gjerne et annet utgangspunkt, nemlig at vi har *gitt* en setning, og ønsker å sjekke om den er med i språket. Vi tenker oss da at automaten er en maskin som hele tiden er i en tilstand, og som leser setningen symbol for symbol. Når det leses et visst symbol skal automaten går fra den aktuelle tilstand, langs en kant merket med dette symbolet, til en ny tilstand. Tilstandsoverganger som tilsvarer ϵ-kanter kan maskinen gjøre spontant når "den får lyst". Merk at det godt kan hende at automaten må stoppe lesingen midt i en setning fordi ingen lovlig tilstandovergang kan gjøres.

Om vi tenker oss en gitt setning, er vår automat "ikkedeterministisk" på to måter: For det første kan den gjøre ϵ-overganger når den får lyst, uavhengig av hva neste symbol i setningen er. For det andre kan det godt være flere kanter ut fra samme tilstand som er merket med neste symbol i setningen, og da kan den velge fritt mellom disse.

Sett fra denne synsvinklen er det rimelig å si at denne maskinen *godkjenner* en setning, dersom det finnes en måte å gjøre de "ikkedeterministiske valg" på som gjør at maskinen er i en sluttilstand når alle symbolene i setningen er lest.

3.2 Deterministiske automater

Som utgangspunkt for en algoritme som skal bestemme om en streng er med i språket eller ikke er de ikkedeterministiske automater (eller deres nære slekninger syntaksdiagrammene) langt fra ideelle. For dette formålet kan man i stedet ønske seg automater der alle tilstandsoverganger er entydig bestemt ut fra den tilstanden de er i og det neste symbolet i setningen.

Vi skal kalle de automater som oppfyller dette kravet for *deterministiske automater*. Det store spørsmålet blir da om vi klarer å beskrive en like stor klasse språk med slike automater som med de mer generelle ikkedeterministiske automater. Som vi har antydet over (ved at begge typer kjenner igjen de regulære språk) er svaret her ja. Vi får til og med en verdifull ting med på kjøpet: Om vi har gitt en ikkedeterministisk automat, så er det en helt kurant affære å konstruere en deterministisk automat som kjenner igjen nøyaktig det samme språket.

For å spare plass skal vi i det følgende snakke om ID-automater og D-automater, i stedet for ikkedeterministike og deterministiske automater. Som man kan tenke seg er den nøyaktige definisjonen på en D-automat at det er *en ID-automat der det (i) ikke finnes ϵ-kanter, og (ii) der det aldri går flere kanter med samme grunnsymbol ut fra samme tilstand.*

Vi ser at automaten i eksempel (E) over tilfredstiller dette kravet. Det å bruke en slik automat til å avgjøre om en gitt setning er med i språket, er selvfølgelig en helt deterministisk prosess. Vi ser også at det er svært lett å implementere denne prosessen som et program til en datamaskin.

3.3 Fra en ikkedeterministisk til en deterministisk automat

Vi skal nå se på hvordan vi ut fra en gitt ID-automat kan lage en D-automat som gjenkjenner nøyaktig det samme språket. Vi skal ikke her gå fullstendig inn i teorien omkring dette (selv om den vil ligge nokså i dagen ut fra de konstruksjonene vi skal gjøre), men skal vise hvordan en slik D-automat kan lages, samt referere en viktig sats i denne forbindelse.

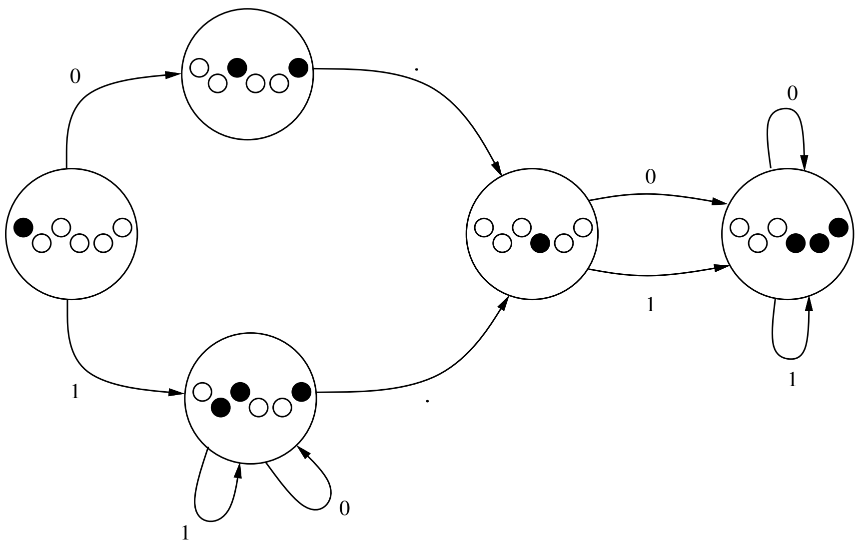
Anta derfor at vi har en ID-automat, og la oss kalle tilstandene i denne for ID-tilstander. Vi lager da en D-automat på følgende måte:

Hver tilstand i D-automaten (senere kalt D-tilstander) skal være identifisert med et utplukk (eller en delmengde) av ID-tilstandene. Det kan være greit å tenke seg at vi i utgangspunktet lager en D-tilstand for hvert eneste slikt utplukk, men vanligvis vil det bare være en lite antall av disse som blir aktuelle fordi de øvrige ikke kan nås fra starttilstanden (som vi skal definere om et øyeblikk).

For å lette beskrivelsen definerer vi "tillukningen" av et utplukk av ID-tilstander. Denne lager vi rett og slett ved å utvide det opprinnelige utplukket med de ID-tilstander man kan komme til ved bare å følge ϵ-kanter. Starttilstanden i D-automaten skal være den som svarer til tillukningen av (mengden av) starttilstanden i ID-automaten. Slutt-tilstandene i D-automaten er nøyaktig de D-tilstandene der det tilsvarende utplukket inneholder minst en av sluttilstandene i ID-automaten.

La oss så feste oss ved en D-tilstand DT og et grunnsymbol g, og så se på hvordan vi skal avgjøre om det skal gå en kant fra denne tilstanden merket med dette grunnsymbol, og i så fall hvilken D-tilstand den skal gå til. Man ser da først på hver av ID-tilstandene i det utplukket som svarer til DT, og samler opp nøyaktig *de ID-tilstander man kan komme til fra disse ved kanter merket med* g. Dersom denne mengden er tom, skal det ikke gå noen g-kant fra DT, ellers tar vi tillukningen av denne mengden og har derved det utplukket som angir den D-tilstanden som g-kanten skal gå til.

Når man skal konstruere en slik D-automat lønner det seg å starte med starttilstanden i D-automaten, og så arbeide seg utover langs de forskjellige kanter. Da behøver men bare skrive opp de D-tilstander man faktisk får bruk for. Om vi nå gjør dette med ID-automaten i eksempel (D) over, ser vi at vi faktisk får den D-automaten som er gjengitt i eksempel (E). For å tydeliggjøre denne konstruksjonen lager vi en forstørret versjon av denne D-automaten der vi i hver D-tilstand har tegnet en skisse av ID-automaten (utenom kantene, men det skulle likevel være mulig å identifisere ID-tilstandene ut fra plasseringen). De utfylte sirklene angir hvilket utplukk av ID-tilstander den enkelte D-tilstand tilsvarer.



Vi skal til slutt gjengi et lemma som vi kaller *Endelig-automat-lemmaet.* Dette lemmaet kan også sees som den invariant som gjelder når vi leser en setning ved hjelp av en D-automat som er konstruert fra en ID-automat slik vi beskrev over. Det kan derfor være grunnpillaren i et bevis for at denne konstruksjonen er riktig.

Endelig-automat-lemmaet:

> *Anta at en D-automat er konstruert fra en ID-automat slik som angitt over, og at denne D-automaten har lest en viss sekvens av grunnsymboler, og dermed kommet i en viss tilstand* DT. *Da vil det utplukk av ID-tilstander som tilsvarer* DT *være de og bare de ID-tilstander som ID-automaten kunne være i etter å ha lest denne sekvensen av grunnsymboler.*

Det kan ofte være lettere å tenke ut fra at ikkedeterministiske automater, i likhet med syntaksdiagrammer, normalt brukes til å *generere* setninger. Ut fra denne vinklingen kan siste delen av lemmaet formuleres: "Da vil det utplukk av ID-tilstander som tilsvarer DT være de og bare de ID-tilstander som ID-automaten kunne være i etter å ha *generert* denne sekvensen av grunnsymboler."

3.4 Litt om leksikalsk analyse

Som nevnt i innledningen er kompilatorer gjerne utstyrt med en preprosessor som leser den rå programteksten tegn for tegn og deler den opp i en sekvens av leksikalske enheter (nøkkelord, identifikatorer, operatorer etc.). Hver enkelt av disse leksikalske kategoriene kan vanligvis beskrives som regulære språk og dermed gjenkjennes av endelige automater.

En leksikalsk analysator vil stort sett kunne konstrueres etter følgende mønster. For hver leksikalsk kategori konstrueres det en (gjerne ikkedeterministisk) endelig automat som kjenner igjen enheter av denne kategorien. Disse automatene settes så sammen til én stor ID-automat ved å innføre en ny starttilstand og ϵ kanter til starttilstandene i de enkelete automatene. Den nye ID-automaten konverteres så til en D-automat etter oppskriften gitt over. Denne D-automaten kan så implementeres på en passelig måte, f.eks. ved hjelp av en tabell.

I den ferdige D-autometen bør det også være slik at den sluttilstanden man kommer til også entydig angir hva slags leksikalsk enhet man nå har lest, altså om det var navn, tall, '<=' osv. Hvordan dette og en del andre detaljer spiller seg ut overlater vi til leseren å tenke over.

En ting man her må passe på er at automaten må lese lengst mulig før den aksepterer en tegnstreng som en leksikalsk enhet. Ta for eksempel tegnsekvensen "`teller2`". Den vanlige tolkningen av denne sekvensen er som navnet "teller2", men hvis automaten ikke leser lengst mulig kunne den leksikalske analysatoren f.eks. splitte dette opp i navnet "teller" og tallet to. Automaten må altså lages slik at selv om den er i en sluttilstand, skal den fortsette å lese dersom også det neste symbolet vil føre til en lovlig overgang.

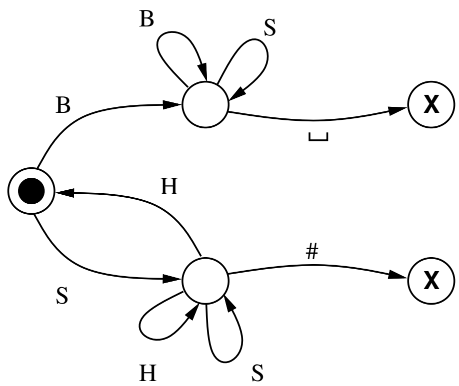
Det finnes systemer som genererer leksikalske analysatorer. LEX og FLEX er to kjente slike systemer. Dette systemet tar som input regulære uttrykk som beskriver de leksikalske kategoriene, samt programbiter som angir hva som skal gjøres når en leksikalsk enhet gjenkjent. Systemet produserer en tabell som svarer til den endelige automaten nevnt over.

Vi avslutter dette kapittelet med et eksempel på gjenkjenning av variabelnavn og heksadesimale tall, begge skrevet med små bokstaver. Variabelnavn må begynne med en bokstav og kan deretter bestå av bokstaver og siffer om hverandre og avsluttes med et blankt tegn. Heksadesimale tall er sekvenser av bokstavene a, b, c, d, e, f og siffer og ender alltid med tegnet '#'. Det er her altså aldri noen tvil om når den leksikalske enheten er slutt.

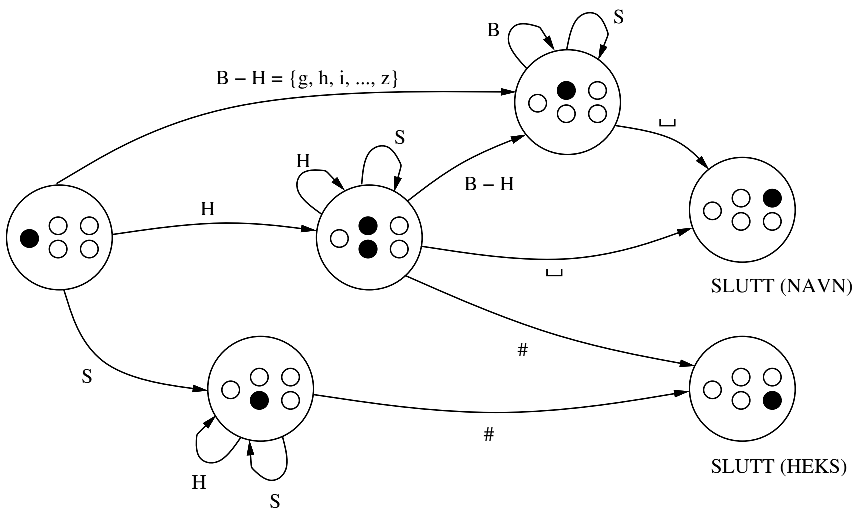
La $B = \{a, \ldots, z\}$, $H = \{a, b, c, d, e, f\}$ og $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Variablenavn og heksadesimale tall kan beskrives med de følgene regulære uttrykk:

$$\begin{array}{lcl} \text{NAVN} & \rightarrow & B\{B \mid S\}^* \sqcup \\ \text{HEKS} & \rightarrow & \{H \mid S\}^+\# \end{array}$$

En ID-automat som gjenkjenner navn og heksadesimale tall kan nå konstrueres rett fram fra de regulære uttrykkene. I ID-automaten under skal en kant som er merket med en symbolmengde (f.eks. B) egentlig forstås som en bunt med parallelle kanter, hver merket med ett av symbolene i mengden:



Den tilsvarende D-automaten konstrueres etter oppskriften gitt tidligere i kapittelet. De vekselsvis tomme og fylte sirklene inne i tilstandene angir som tidligere hvilket utplukk av ID-tilstander hver D-tilstand tilsvarer.



Kapittel 4

Parsering, og algoritmer for parsering

Som vi så i forrige kapittel vil altså en kompilator starte med å omforme det programmet den holder på med til en sekvens av leksikalske enheter (navn, tall, spesialtegn, de forskjellige nøkkelord osv.), og dette gjøres vanligvis i en enkel preprosessor ("leksikalsk analyse").

Neste oppgave vil så være å avgjøre om den sekvensen som da framkommer utgjør et riktig program. Vi er imidlertid interessert i mer enn et ja/nei-svar på dette. Dersom programmet er riktig er kompilatoren også ute etter den "syntaktiske struktur" av programmet, som altså grovt sett vil fortelle hva slags konstruksjoner (deklarasjon, if-setning osv) det er snakk om, og hvor disse starter og slutter. Grunnen til at dette er viktig er selvfølgelig at meningen med et program vanligvis er nært knyttet til den syntaktiske strukturen, og at denne derfor er sentral når programmet skal oversettes videre til maskinkode. Denne siste delen av en kompilators arbeid, den såkalte *kodegenerering*, skal vi imidlertid ikke se på i dette kompendiet. (Den blir behandlet i et hovedfagskurs, og noe i IN-102).

Vi skal i det følgende anta at de grammatikkene vi ser på er rene BNF-grammatikker, og at de også, om ikke annet er sagt, er entydige. Det å spørre etter den syntaktiske strukturen til et (syntaktisk riktig) program kan dermed konkretiseres til å spørre etter det entydig definerte syntakstreet for dette programmet. Bladene i dette treet er altså de leksikalske enhetene som programmet er bygget opp av.

Det å konstruere syntakstreet for en gitt setning kalles gjerne *parsering*, og algoritmer som er utviklet for dette kalles derved *parserings-algoritmer*. Deres oppgave er altså først og fremst å avgjøre om det aktuelle program er en setning i språket, og i så fall komme opp med syntaksstreet. Dersom programmet *ikke* er riktig (hvilket det som kjent ofte ikke er) kan man også ønske at kompilatoren kan si noe om hvor feilen er og hva den består i.

4.1 Innledende armøvelser

Før vi starter å se på selve de algoritmene som er aktuelle, og teorien rundt dem, skal vi gjøre litt generelle forberedelser. Om ikke annet er sagt skal vi altså anta at vår grammatikk er en entydig, vanlig BNF-grammatikk.

4.1.1 Representasjon av den syntaktiske struktur

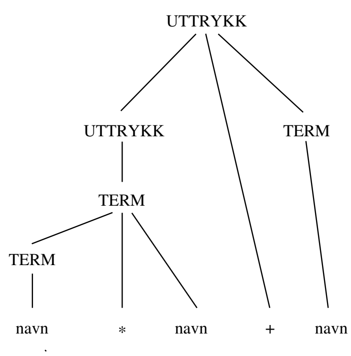
Man kan tenke seg syntakstreet for en setning representert på mange forskjellige måter. Det letteste er kanskje å tenke seg det satt opp med (Simula-aktige) objekter som noder og pekere som kanter, og ofte er dette også brukt i praksis. For store programmer tar dette imidlertid mye plass, og i praksis forsøker man derfor ofte bare å lagre de delene av treet man får behov for siden, og ellers kaste mest mulig etter hvert.

En annen måte, som kanskje først og fremst kan være behagelig å *tenke* ut fra, er at vi tenker oss trestrukturen markert som "syntaksparenteser" i den opprinnelige setningen. Dette vil si at vi for hver node i treet lager en syntaksparentes omkring den konstruksjonen som er avledet fra denne noden, og at vi merker hver slik parentes med den produksjonen som er gjort i noden.

Vi skal skrive slike syntaksparenteser som vanlige parenteser, men med et heltall knyttet til, som angir hvilken produksjon de tilsvarer. Vi skal altså tenke oss at produksjonene er nummerert. Som et eksempel kan vi se på en grammatikk vi så på tidligere, og vi gjengir først denne, med nummere på produksjonene

$$
\begin{array}{lcll}
\text{UTTRYKK} & \rightarrow & \text{UTTRYKK} + \text{TERM} & (1) \\
\text{UTTRYKK} & \rightarrow & \text{TERM} & (2) \\
\text{TERM} & \rightarrow & \text{TERM} * \textbf{navn} & (3) \\
\text{TERM} & \rightarrow & \textbf{navn} & (4)
\end{array}
$$

Et tradisjonelt syntakstre for setningen "**navn** $*$ **navn** $+$ **navn**" kan her se slik ut:



Om vi skriver ut den samme syntaktiske strukturen med syntaksparenteser, tar den seg slik ut:

$$({}_1({}_2({}_3({}_4\ \textbf{navn}\ {}_4)\ *\ \ \textbf{navn}\ {}_3)\,{}_2) + ({}_4\ \textbf{navn}\ {}_4)\,{}_1)$$

En måte å se parseringens problem på, er at man skal finne fram til syntaksparentesene, som ligger "skjult" mellom grunnsymbolene i den opprinnelige setningen. Om vi først har dem, er det selvfølgelig ingen problemer med f.eks. å bygge opp treet. Den rekkefølge vi forsøker å finne disse parentesene i, tilsvarer forskjellige strategier for parsering.

Den strukturen som angis av syntaksparentesene kalles ofte *frasestrukturen* i setningen, og det som ligger inne i en parentes kalles en *frase*. Vi kan også snakke om frasestrukturen i en setningsform (altså, en halvavledet setning). Vi kan f.eks. angi at setningsformen "TERM $*$ **navn** + TERM" har følgende frasestruktur:

$$({}_1({}_2({}_3\ \mathrm{TERM} * \mathbf{navn}\ {}_3)\,{}_2) + \ \mathrm{TERM}\ {}_1)$$

Det som i en setning eller setningsform ligger inne i en "innerste" syntaksparentes, kalles en grunnfrase. "TERM $*$ **navn**" er eneste grunnfrase i setningsformen over.

For lettere å kunne uttrykke oss, skal vi i det følgende ofte snakke om "et metasymbols høyresider". Med dette skal vi mene høyresidene i de produksjoner som har dette metasymbolet på venstresiden.

På liknende måte skal vi si at en syntaksparentes *stammer fra et visst metasymbol*. Dette skal bety at den produksjonen som syntaksparentesen tilsvarer har dette metasymbolet som sin venstreside, hvilket igjen vil si at den frasen denne parentesen omslutter er avledet fra dette metasymbolet.

4.1.2 Parsering "ovenfra-ned" og parsering "nedenfra-opp"

Når man skal forsøke å finne den syntaktiske strukturen i en setning, vil man som regel velge enten å starte med å lete etter de *ytterste* syntaksparentesene først, eller med de *innerste* først. Det å lete etter de ytterste først, svarer til å starte i roten av syntakstreet, og å forsøke å avlede noe fra startsymbolet som til slutt blir den setningen som vi skulle finne strukturen i. Vi vil da underveis ha en setningsform, og denne vil vi hele tiden kjenne frasestrukturen i, siden vi selv har avledet den. Steget blir å velge et passelig metasymbol i den halvferdige setningsformen, og å velge en høyreside for dette som passer med den setningen vi har.

Vi forsøker her altså å gjøre en avledning fra startsymbolet mot den gitte setningen. Siden vi starter i roten (og siden data-trær vokser nedover!) kalles denne strategien "ovenfra–ned" ("top–down").

Den omvendte strategien er å forsøke å finne grunnfraser i den foreliggende setning, altså lete etter "innerste syntaksparenteser". Man kunne f.eks. starte med å sette (4–4) parenteser rundt hvert 'navn' som ikke har symbolet '$*$' foran seg. (Det lar seg vise at dette må være riktig i den grammatikken vi anga over).

I stedet for å sette inn selve syntaksparentesene vil man imidlertid ofte løpende erstatte den frasen de omslutter med metasymbolet på venstre side i den tilsvarende produksjonen. En slik substituering av en grunnfrase med det tilsvarende metasymbol kalles en *reduksjon*, og strategien her blir da å forsøke å gjøre suksessive reduksjoner, inntil man bare står igjen med grunnsymbolet. Av tilsvarende grunner som over kalles denne strategien "nedenfra–opp" ("bottom–up").

Enda et par betegnelser som gjerne brukes om dette: Nedenfra-opp parsering kalles også "reduktiv parsering" eller "reduktiv syntaksanalyse". Tilsvarende kalles ovenfra-ned parsering for "produktiv syntaksanalyse".

4.1.3 Parsering med én gangs gjennomlesing

Når man i praksis skal parsere en setning er det vanligvis ønskelig at man bare leser gjennom (symbolene som utgjør) setningen én gang (og i vår vestlige kultur: fra venstre mot høyre). Man må da passe på at man får gjort all avledning eller reduksjon i forbindelse med denne ene gjennomgangen, og det viser seg da at to strategier naturlig peker seg ut. Den første er å forsøke å gjøre de aktuelle operasjoner når man under lesing mot høyre passerer venstresyntaksparenteser, og den andre at man forsøker å gjøre dem når man passerer de tilsvarende høyresyntaksparenteser.

Vi skal i det følgende ofte få bruk for å snakke om det symbolet som er det neste til å bli lest, og dette skal vi kalle *klarsymbolet*. I praksis vil lesemekanismen ligge ett hakk foran det vi tenker oss i våre teoretiske betrakninger, og klarsymbolet vil hele tiden ligge til alminnelig beskuelse i en spesiell variabel.

For lettere å ha kontroll med slutten av setningen vil vi ofte legge på et eget sluttmerke '@' for setningene, ved å utvide grammatikkene med en produksjon "S' $\rightarrow$ S @", der S er det gamle startsymbolet. Vi forutsetter da at hverken S' eller '@' er brukt i grammatikken tidligere. En fordel vi også får av dette er at det nye startsymbolet S' ikke vil forekomme i noen av høyresidene, og dette kan ofte være behagelig. Når vi har lest hele den opprinnelige setningen, vil altså klarsymbolet være '@'.

Sluttsymbolet '@' tjener som et "merke for tekstslutt" og er tenkt satt inn på slutten av alle setninger (både syntaktisk riktige og syntaktisk gale). Hvis teksten som skal analyseres ligger på en fil, vil sluttsymbolet svare til filsluttmerket (EOF).

4.1.4 Tre viktige mengder

Når man skal utforme parserings-algoritmer for en gitt grammatikk, er det tre mengder som ofte kommer inn på forskjellige måter, nemlig:

Startmengden for hvert metasymbol:
Dette er, for hvert metasymbol, mengden av de grunnsymboler som kan stå først i en frase avledet fra dette metasymbolet.

Etterfølgermengden for hvert metasymbol:
Dette er, for hvert metasymbol, mengden av de grunnsymboler som i noen setningsform i grammatikken kan stå etter dette metasymbolet.

Meta-til-tom mengden:
Dette er mengden av de metasymboler som kan videreavledes til den tomme setning.

Disse mengdene er ikke vanskelige å beregne for en gitt grammatikk, ved hjelp av intuitivt forståelige "kompletteringsprosesser". Det som kompliserer det hele litt er de metasymbolene som kan produsere den tomme setning, og derfor er det behagelig å beregne meta-til-tom mengden først. Det gjøres som følger:

Initialisering: La meta-til-tom mengden bestå av de metasymboler som har en tom høyreside.

Steg: Dersom det finnes et metasymbol utenfor meta-til-tom mengden som har en høyreside som bare består av metasymboler som allerede er med i mengden, så inkluder dette i mengden.

Komplettering: Gjenta steget inntil meta-til-tom mengden ikke øker mer.

Når så denne er beregnet, kan vi beregne start-mengdene. For lettere å beskrive prosessen, skal vi også si at grunnsymboler har en startmengde, nemlig seg selv. Beregningen av startmengdene må gjøres kollektivt, for alle metasymbolene på en gang:

Initialisering: La startmengden for hvert av metasymbolene være de grunnsymbolene som står først i en høyreside for dette metasymbolet. (Denne kan være tom for noen av metasymbolene).

Steg: Se på en høyreside for et metasymbol M. For hvert (grunn- eller meta-) symbol x i høyresiden som er slik at det enten står først, eller det bare står metasymboler i meta-til-tom mengden foran det, så øker vi startmengden til M med startmengden til x.

Komplettering: Gjenta steget til ingen av startmengdene kan økes mer.

Til slutt kan vi beregne etterfølgermengdene kollektivt slik:

Initialisering: For hvert metasymbol, la etterfølgermengden være unionen av startmengdene til de (grunn- eller meta-) symboler som i en eller annen høyreside enten står *umiddelbart bak* dette metasymbolet, eller bak det slik at *de mellomliggende symboler er metasymboler i meta-til-tom mengden.*

Steg: Se på en forekomst av et metasymbol M i en høyreside til metasymbolet N, slik at M enten står bakerst, eller det bak M bare står metasymboler i meta-til-tom mengden. Øk etterfølgermengden til M med etterfølgermengden til N.

Komplettering: Gjenta steget inntil ingen av etterfølgermengdene kan økes mer.

Det er ikke vanskelig å vise at disse algoritmene må virke riktig, men vi skal ikke gå inn på detaljene her. Hovedlinjen i bevisene er først å tenke seg en situasjon der (om vi tar den første algoritmen) et metasymbol avledes til den tomme setningen, og ut fra det overbevise seg om at da må dette metasymbolet til slutt tas inn i meta-til-tom mengden. Deretter må man anta at et gitt metasymbol kommer med i meta-til-tom mengden ut fra algoritmen over, og ut fra dette konstruere en avledning fra dette metasymbolet som faktisk gir den tomme mengden. Den rekkefølgen som algoritmen tok symbolene inn i mengden, blir her viktig.

I stedet skal vi titte på et par eksempler, og vi ser først på vår gamle grammatikk:

UTTRYKK'	$\rightarrow$	UTTRYKK @
UTTRYKK	$\rightarrow$	UTTRYKK + TERM
UTTRYKK	$\rightarrow$	TERM
TERM	$\rightarrow$	TERM $*$ **navn**
TERM	$\rightarrow$	**navn**

Vi konstaterer først at meta-til-tom-mengden er tom. Videre får vi følgende mengder (det som står til venstre for den loddrette streken, er det som kom inn under initialiseringen):

Metasymbol	Startmengde		Etterfølgermengde	
UTTRYKK		**navn**	@ +	
TERM	**navn**		$*$	@ +

Vi ser så på en annen grammatikk, som angir det samme språket, men som er bedre tilpasset såkalt LL-parsering (som vi skal se på om et øyeblikk). Vi bruker her litt forkortede navn på metasymbolene:

$$
\begin{array}{lll}
\text{U'} & \rightarrow & \text{U @} \\
\text{U} & \rightarrow & \text{T UX} \\
\text{UX} & \rightarrow & \epsilon \mid + \text{ T UX} \\
\text{T} & \rightarrow & \textbf{navn} \text{ TX} \\
\text{TX} & \rightarrow & \epsilon \mid * \textbf{ navn} \text{ TX}
\end{array}
$$

Her ser vi lett at meta-til-tom mengden består av UX og TX. Videre får vi, etter samme skjema som over:

Metasymbol	Startmengde		Etterfølgermengde	
U		**navn**	@	
UX	+			@
T	**navn**		+	@
TX	$*$			+ @

Øvelse

Som en øvelse kan man her, spesielt for den siste grammatikken, lage setningsformer der hver av de forskjellige etterfølgerrelasjonene er oppfylt.

4.2 LL-parsering

Den mest "optimistiske" strategien for parsering er å forsøke å identifisere alle venstresyntaksparenteser (inklusive deres nummer) så fort man kommer til dem under lesingen av setningen. En forutsetning er da at vi hele tiden får lov til å titte på klarsymbolet, men det er likevel ikke alle grammatikker der slik bestemmelse er mulig. Om det i utgangspunktet ikke er mulig, lar det seg ofte gjøre etter å ha vridd litt på grammatikken på passelige måter. Vi skal titte litt på dette lenger ned.

Det å utføre parsering etter dette prinsippet kalles *LL-parsering*, og den mest kjente måten å administrere en slik parsering på er den såkalte "recursive descent" metoden. Den første L-en står for at vi leser *fra venstre* mot høyre, mens den andre betyr at vi satser på å finne *venstre*enden av hver frase (syntaksparentes). Vi skal siden se på det som tilsvarende kalles LR-parsering.

For å ha håp om kunne avgjøre hva slags syntaksparenteser som skal være mellom det siste symbolet vi har lest og det neste, må vi altså hele tiden hvertfall kunne se ett symbol framover, alstå se på klarsymbolet (ellers kunne språket bare ha en setning!). De grammatikker der det er teoretisk mulig å bestemme hvilke venstre syntaksparenteser som starter der vi nå står ved bare å titte ett symbol framover, kalles *LL(1)-grammatikker*. Man kan også tenke seg å kunne titte flere symboler framover, og generelt bruker vi følgende (noe mer statiske) definisjon:

Definisjon av LL(k)-grammatikker

> En grammatikk er LL(k) dersom man, for alle setninger i det genererte språket, og for enhver oppdeling av en slik i to deler, alltid har nok informasjon til å bestemme alle venstre-syntaksparenteser i den venstre delen (inklusive de mellom høyre og venstre del) ved bare å se på den venstre delen samt de k første symbolene i den høyre delen. Merk at oppdelingspunktet kan være helt forrest eller helt bakerst i setningen. Om den høyre delen er kortere enn k symboler skal man tenke seg setningen fylt ut med et passelig antall avslutningssymboler.

Vi kan observere at den grammatikken som er gitt for UTTRYKK tidligere, ikke er LL(1). F.eks. kan vi i setningen "**navn** $*$ **navn** $+$ **navn**" umulig avgjøre om den første parentesen skal være en '$(_1$' eller en '$(_2$' før vi har lest det meste av setningen. Dette avhenger nemlig av om det i det hele tatt kommer noen '+' senere i setningen. Det er imidlertid ikke vanskelig å lage en LL(1)-grammatikk for det samme språket, men slik den her er skrevet ser vi at den faktisk ikke er LL(k) for noen fast k.

Rent intuitivt (og svært løslig) kan man si at et språk har *LL(1)-natur* dersom det hver gang vi går inn i en setningskonstruksjon (ved lesing fra venstre mot høyre) er mulig å avgjøre hva slags konstruksjon dette er allerede ut fra første symbolet. Typiske LL(1)-konstruksjoner er derfor de som starter med et eget nøkkelord, så som f.eks. if-setninger og variabel-deklarasjoner i Simula. Legg imidlertid merke til at denne betrakning er svært avhengig av *hva* vi regner som "samme type konstruksjon", noe som igjen er avhengig av hvordan grammatikken er satt opp.

Som en avsluttende eksempel her kan det sies at sentrale deler av Simulas syntaks ut fra dette kan sies å ha LL(2)-natur. Tenk deg f.eks. at du nettopp har lest et semikolon etter en deklarasjon, og at du ser at klar-symbolet er et navn. Hva slags konstruksjoner kan dette navnet være starten på, og hvordan kan du skille mellom disse alternativene?

4.2.1 Algoritmer for LL-parsering

Når vi behandler syntaksparentesene i den rekkefølge deres venstre-syntaksparenteser kommer, vil vi hele tiden behandle de ytre parentesene før de som ligger lenger inn, og ut fra det som er diskutert tidligere, er LL-parsering dermed en "ovenfra–ned" metode.

Om man tenker seg strukturen for den setningen som skal parseres lagt ut som et syntakstre ser vi videre at for noder på samme nivå, vil man med LL-parsering ta de til venstre før de til høyre. Husk altså at hver indre node i syntakstreet tilsvarer en syntaksparentes (rundt nodens subtre). Nodene vil dermed totalt bli behandlet i *prefiks* rekkefølge. Om vi tenker oss det hele administrert som en avledning fra startsymbolet (slik vi diskuterte for ovenfra-ned parsering tidligere), ser vi at dette vil tilsvare en venstreavledning, altså slik at vi hele tiden utvikler videre det metasymbolet som ligger lengst til venstre.

Vi skal nå konkretisere nærmere hva det innbærer at en grammatikk er LL(1). I tillegg til startmengden for et *metasymbol* som vi har definert før, får vi nå også bruk for å snakke om *startmengden til en produksjon*. Dette er per definisjon de grunnsymboler som kan stå først i en setning avledet fra *høyresiden* i denne produksjonen. Vi vil også snakke om startmengden til en *høyreside* i den opplagte betydning.

For enkelhets skyld skal vi først se på den type grammatikker der ingen av produksjonene har tomme høyresider (og der meta-til-tom-mengden dermed er tom). Det er da svært enkelt å bestemme startmengden til hver av grammatikkens produksjoner, den er da rett og slett identisk med startmengden til det første (meta- eller grunn-) symbolet i høyresiden av produksjonen. Husk her at startmengden til et grunnsymbol er (mengden av) symbolet selv.

For en slik grammatikk kan kriteriet for om den er LL(1) formuleres ganske enkelt:

En grammatikk uten tomme høyresider er LL(1) hvis og bare hvis:
For hvert metasymbol er startmengdene til de alternative høyresidene disjunkte.

For å bevise denne satsen inngår to ting. Det ene er å vise at dersom grammatikken ikke tilfredstiller dette kravet, så er den ikke LL(1). Man må da vise at det finnes en setning i det genererte språket og en oppdeling av denne, som ikke tilfredstiller kravet for LL(1)-grammatikker. Dette følger nokså greit fra negasjonen av kriteriet i setningen over, og vi skal ikke gå ytterligere inn på dette.

Det andre er å vise at dersom en grammatikk tilfredstiller kravet over, så er det alltid mulig løpende å bestemme alle venstresyntaksparentesene fram til klarsymbolet. Dette skal vi vise ved å angi en algoritme som klarer denne jobben dersom kriteriet i satsen over er oppfylt.

En slik algoritme kan organiseres på flere måter. Én måte er den som kalles "prediktiv parsering", men den vi skal se på her er den som kalles "recursive descent". Mange vil ha vært borte i denne før (dog ut fra syntaksdiagrammer, men vi skal se på det om litt). Ideen bak metoden er at man for hvert metasymbol skriver en prosedyre, som skal kalles når vi vet at det på det stedet vi nå er i parseringen (altså foran klarsymbolet) skal være en venstresyntaksparentes som stammer fra dette metasymbolet (uten at vi vet hvilken alternativ høyreside som er den riktige).

Det første en slik prosedyre skal gjøre er å se på klarsymbolet, og på grunnlag av dette avgjøre hvilken høyreside som skal velges. At dette er mulig følger greit av kriteriet i satsen. Siden ingen metasymboler kan produsere den tomme streng, må klarsymbolet nettopp være det første symbolet i noe som er avledet fra den aktuelle høyresiden, og det må dermed være med i startmengden for denne høyresiden. Men da disse startmengdene er disjunkte, er det bare én som kan være aktuell, og man velger denne. (Dersom klarsymbolet ikke er med i noen av mengdene må det være noe galt med den setningen vi leser).

Dermed får altså prosedyren bestemt hvilken høyreside som er den riktige, og den konsentrerer seg så om denne og behandler den symbol for symbol. Når det aktuelle symbol er et grunnsymbol vil den rett og slett lese neste symbol fra setningen, og sjekke at det nettopp er dette symbolet som kommer (noe det må være om det er en riktig setning).

Når symbolet i høyresiden er et metasymbol vil algoritmen vite at det på dette stedet i setningen (altså foran klarsymbolet) må være en venstresyntaksparentes som stammer fra dette metasymbolet. Dermed står den i posisjon til å kalle prosedyren for dette metasymbolet, og gjør rett og slett det. Denne vil så på samme måten først bestemme hvilke av de aktuelle alternativene som er det riktige, osv.

Det hele startes opp ved å kalle den prosedyren som tilsvarer startsymbolet. Vi kan nå lett overbevise oss om at hver prosedyre her vil lese og sjekke en del av setningen som utgjør en frase som kan avledes fra det aktuelle metasymbol, og at den gjør det på den eneste mulige måte (så lenge kriteriet i satsen gjelder). Det vil si at dersom vi med disse prosedyrene leser en setning i språket, så må den bli parsert riktig.

Vi kan også observere at vi med denne metoden bestemmer syntaksparentesene og deres nummer i to steg: Først bestemmer den omgivende prosedyre at her må det være en syntaktisk parentes som stammer fra et visst metasymbol. Dermed kalles den tilsvarende prosedyren og det første denne gjør er å bestemme nøyaktig hvilken av de alternative høyresidene som skal velges.

Det er en komplikasjon man her skal være klar over, nemlig at dersom grammatikken har direkte eller indirekte venstrerekursjon, så kunne det tenkes at den angitte algoritmen fortsetter å gjøre prosedyrekall uten noen gang å lese neste symbol. Imidlertid vil dette ikke inntreffe, da grammatikker som har venstrerekursjon aldri vil kunne tilfredstille kriteriet i satsen. Det alternativet som har venstrerekursjon vil nemlig da ha overlappende startmengde med alle de andre alternative høyresidene.

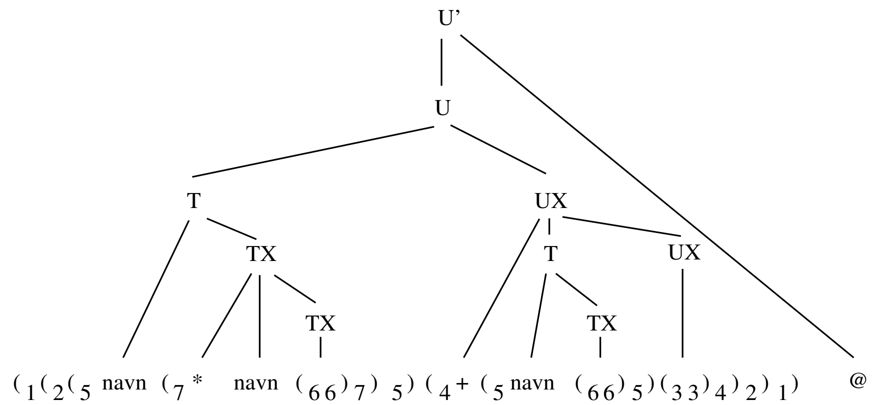
Dermed skulle satsen over være vist.

Grammatikker med tomme høyresider

Nå er det dessverre slik at svært mange av de grammatikker det er aktuelt å gjøre LL(1)-parsering for, har tomme høyresider, og da må vi være litt forsiktige. Hvis en slik tom høyreside er valgt i avledningen av en setning vil den tilhørende syntaksparentesen også være tom, og klarsymbolet vil da være det grunnsymbolet som kommer bak denne igjen. Klarsymbolet vil altså være med i etterfølgermengden til det metasymbolet som syntaksparentesen stammer fra. Eksemplet under viser en slik situasjon. Grammatikken som brukes er den modifiserte grammatikken fra side 26:

U'	$\rightarrow$	U @
U	$\rightarrow$	T UX
UX	$\rightarrow$	ϵ
UX	$\rightarrow$	+ T UX
T	$\rightarrow$	**navn** TX
TX	$\rightarrow$	ϵ
TX	$\rightarrow$	$*$ **navn** TX

Setningen "**navn** $*$ **navn** $+$ **navn**" har følgende syntaktiske struktur:



La situasjonen beskrevet over, være en motivasjon til følgende definisjon:

For å fange inn dette i passelige begreper, vil vi definere en ny og utvidet variant av etterfølgermengden for en produksjon. Merk her at ikke bare høyresiden, men også metasymbolet på venstresiden, står sentralt i definisjonen.

Den utvidede startmengden til en produksjon:

For en gitt produksjon består denne mengden av alle symbolene i startmengden til de symbolene på høyresiden som enten står først eller bare har metasymboler i meta-til-tom mengden foran seg. Hvis høyresiden bare består av metasymboler i meta-til-tom mengden, inkluderes også symbolene i etterfølgermengden til produksjonens venstreside.

Vi ser at den utvidede startmengden til en produksjon inneholder nøyaktig de symboler som kan forekomme som klarsymbol, når vi er ved starten av en syntaksparentes for denne produksjonen. Det generelle LL(1)-kravet kan da formuleres som følger:

Det generelle LL(1)-krav:

En grammatikk er LL(1) hvis og bare hvis de produksjonene som har et gitt metasymbol som venstreside, alltid har disjunkte utvidede startmengder.

Legg merke til at dette spesielt vil si at maksimalt én av høyresidene for hvert metasymbol kan produsere den tomme setningen. Om det er flere, vil nemlig alle disse ha etterfølgermengden til venstresiden i sin utvidede startmengde, og disse er derfor ikke disjunkte.

Dersom dette kravet er oppfylt ser vi imidlertid at vi uten videre kan skrive en recursive descent parser for den aktuelle grammatikken på samme måte som vi gjorde over. Når prosedyren for et visst metasymbol skal avgjøre hvilket alternativ som skal velges, ser den bare på hvilken av de utvidede startmengdene klarsymbolet ligger i, og velger det tilsvarende alternativ. Ut fra kravet blir jo dette entydig. Dersom klarsymbolet ikke ligger i noen av de utvidede startmengdene, må den setningen vi ser på være gal.

Studerer vi den siste grammatikken på side 26 (som vi også beregnet meta-til-tom mengden, startmengdene og etterfølgermengdene for) så kan vi nå konstatere at denne er LL(1). Vi kan da også sette opp følgende tabell over hvilket alternativ som skal velges ut fra klarsymbolet for hvert metasymbol. Vi ser lett at dersom vi kommer i en av de de situsjonene som er merket '-', så må det være en feil i setningen.

	navn	+	$*$	@
U	U $\rightarrow$ T UX	-	-	-
UX	-	UX $\rightarrow$ + T UX	-	UX $\rightarrow \epsilon$
T	T $\rightarrow$ **navn** TX	-	-	-
TX	-	TX $\rightarrow \epsilon$	TX $\rightarrow *$ **navn** TX	TX $\rightarrow \epsilon$

Det er en fin øvelse å skrive ut de fire recursive descent prosedyrene for denne grammatikken ut fra denne tabellen. Man må da tenke seg en prosedyre som leser inn neste symbol (og som passelig fyller opp klar-symbol-variabelen på nytt).

Grammatikker som ikke er LL(1), og noen forslag til hjelp

Det er noen problemer som ofte dukker opp når man skal skrive en LL-parser, f.eks. ved recursive descent metoden. Vi skal se på de to vanligste, og hvordan man kan forsøke å gjøre noe med dem. Legg imidlertid merke til at man etter å ha gjort de forbedringene som her foreslås, ikke på noen måte er garantert å få en grammatikk som er LL(1)!

Det første vi skal se på er at grammatikken har venstrerekursjon, slik som f.eks. i den første av følgende produksjoner fra en grammetikk vi har sett på tidligere.

$$
\begin{array}{lll}
\text{UTTRYKK} & \rightarrow & \text{UTTRYKK + TERM} \\
\text{UTTRYKK} & \rightarrow & \text{TERM}
\end{array}
$$

Vi har her også tatt med en alternativ høyreside til UTTRYKK, som ikke er rekursiv. En slik må jo finnes, om man skal kunne lage setninger med bare grunnsymboler. Det er nå lett å innse at denne grammatikken ikke kan være LL(1), rett og slett fordi startmengdene til de to alternativene begge må inkludere startmengden til TERM, slik at de altså umulig kan være disjunkte.

En grei måte å løse dette på er først å skrive disse produksjonene ut som utvidet BNF, på en måte som i det hele tatt ikke inneholder rekursjon. Deretter kan vi forsøke å ta det hele tilbake til ren BNF, med i stedet å bruke høyrerekursjon. Vi får da først følgende:

$$
\text{UTTRYKK} \rightarrow \text{TERM } \{+ \text{ TERM }\}^*
$$

Dette kan vi så skrive ut som følger:

$$
\begin{array}{lll}
\text{UTTRYKK} & \rightarrow & \text{TERM XTERM} \\
\text{XTERM} & \rightarrow & \epsilon \\
\text{XTERM} & \rightarrow & \text{+ TERM XTERM}
\end{array}
$$

Typisk med denne omskrivingen er altså at man både får inn et nytt metasymbol (XTERM), samt at man får en tom høyreside. Ingen av delene er særlig hyggelig, men vi kan jo håndtere det. Vi ble hvertfall kvitt venstrerekursjonen.

Det andre problemet er at flere høyresider til samme metasymbolet ofte kan begynne på samme måten, men ha forskjellige avslutninger. Anta f.eks. at en grammatikk har produksjonene:

$$\begin{array}{lll} \text{SETNING} & \rightarrow & \text{UTTRYKK} + \text{TERM} \\ \text{SETNING} & \rightarrow & \text{UTTRYKK} * \text{TERM} \end{array}$$

Her må selvfølgelig startmengden på de to alternativene bli overlappende, og grammatikken er derfor ikke LL(1). Vi kan her igjen innføre et nytt metasymbol som nå skal stå for den varierende enden. Vi kan da i stedet beskrive dette ved følgende grammatikk:

$$\begin{array}{lll} \text{SETNING} & \rightarrow & \text{UTTRYKK XSETNING} \\ \text{XSETNING} & \rightarrow & + \text{TERM} \\ \text{XSETNING} & \rightarrow & * \text{TERM} \end{array}$$

Denne teknikken kalles ofte “venstre-faktorisering” (vi trekker det som er felles ut til venstre som en “felles faktor”). Rent lokalt er nå denne grammatikken hvertfall blitt LL(1).

4.2.2 Bruk av “recursive descent” ut fra syntaksskjemaer

Vi har til nå snakket om å bruke recursiv descent metoden for grammatikker som er gitt på standard BNF. Det er imidlertid minst like vanlig å bruke denne metoden ut fra at grammatikken er gitt ved syntaksdiagrammer eller ved utvidet BNF. En fordel man da får er at det gjerne blir færre (og større) prosedyrer, som også gir færre prosedyrekall under utførelsen. Det vil vanligvis også bli færre metasymboler (= syntaksdiagrammer) som kan produsere den tomme setning, selv om de fremdeles kan forekomme.

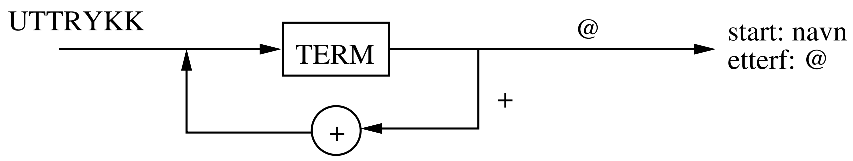
Det er klart at heller ikke alle grammatikker gitt ved syntaksdiagrammer kan føres (direkte) over i recursive descent prosedyrer. Hovedkravet må være at *hver gang man kommer til et forgreningspunkt i diagrammet, må man på grunnlag av klar-symbolet entydig kunne avgjøre hvilken gren man skal velge.* Dersom dette kravet er oppfylt, er det rimelig å si at grammatikken som er gitt ved syntaksdiagrammene er LL(1).

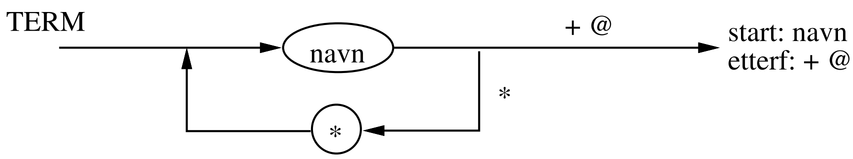
For å undersøke om dette er tilfelle for et gitt sett med syntaksdiagrammer kan vi tenke oss at diagrammene brukes til å *produsere* setninger. For alle forgreningspunkter i diagrammene setter vi så opp for hver utgående gren mengden av de grunnsymboler som kan produseres først dersom man velger denne grenen. Kravet blir da at i hvert forgreningspunkt må de mengdene som er tilordnet de forskjellige utgående grenene, være disjunkte.

For å kunne beregne disse grunnsymbol-mengdene må vi gjøre en tilsvarende beregning som vi gjorde da vi fant startmengdene, etterfølgermengdene og meta-til-tom mengden for vanlig BNF-grammatikker. Vi må altså først beregne hvilke diagrammer som kan produsere den tomme setning, deretter hvilke grunnsymboler hvert skjema kan produsere som første symbol, og til slutt hva etterfølgermengden til hvert skjema vil være. Ut fra skjemaenes utseende vil dette ofte være nokså greie beregninger, og når vi først har gjort dette vil det være kurant å beregne grunnsymbolene som tilsvarer de forskjellige grenene ut fra forgreningspunktene.

Det kan være fristende å sammenlikne et slikt LL(1)-syntaksdiagram med en deterministisk automat, og kanskje kalle det et “deterministisk” syntaksdiagram, i den forstand at om vi skal bruke det til å tolke en gitt setning, så er det hele tiden bare én bestemt vei som er aktuell å gå.

Vi ser til slutt på et meget enkelt eksempel, der ingen av skjemaene kan produsere den tomme setning. Skjemaene er gitt under (UTTRYKK er start-symbol), og de beskriver det samme uttrykksspråket vi har sett på tidligere. Startmengdene og etterfølgermengdene er beregnet og angitt på skjemaene, og vi kan dermed lett sette på de grunnsymbolmengdene som svarer til hver forgrening. Vi ser da at disse skjemaene tilfredstiller LL(1)-kravet for syntaksdiagrammer.





Øvelse

Gjør en tilsvarende analyse av syntaksen for Minila, som man skriver kompilator for i IN-102.

4.3 LR-parsering

Vi skal nå gå over til en helt annen parseringsstrategi, nemlig den at vi i stedet for å forsøke å bestemme venstresyntaksparentesene, satser på å finne *høyre*syntaksparentesene i det øyeblikk vi passerer dem. I stedet for å forsøke å bestemme alt om en frase så fort vi går *inn* i den (slik som i LL-parsering), skal vi altså forsøke å bestemme den fulle frasestruktur i den delen vi legger *bak* oss. Vi ser rent intuitivt at dette er et mer beskjedent mål, og vi kan derfor gjøre oss forhåpninger om at vi med denne strategien kan behandle flere grammatikker, uten fiklete omskrivninger.

Vi understreker at vi i hele omtalen av LR-parsering tenker oss at grammatikken er en ren BNF-grammatikk, og at denne er entydig. Vi skal av bekvemmelighetshensyn også anta at grammatikken ikke har tomme høyresider. Resonnementene vi skal gjennomføre blir da noe enklere, selv om de resultatene vi kommer til faktisk vil gjelde uavhengig av dette. På slutten av dette kapitlet vil vi knytte noen kommentarer til forekomst av tomme høyresider, og vi skal også se litt på hva som skjer om grammatikken ikke er entydig (det kan faktisk av og til utnyttes til vår fordel!).

Diverse måter å beskrive parseringsrekkfølgen ved LR-parsering

Ut fra den samme filosofien som for LL-betegnelsen, er det rimelig å kalle dette LR-parsering da vi leser fra venstre, men bestemmer høyre ende av hver frase. Vi ser at vi ut fra denne strategien alltid kommer til å behandle indre parenteser før de som ligger utenpå, og vi har derved en nedenfra-opp metode. Om vi ser det i forhold til syntakstreet, ser vi at noder (som altså tilsvarer syntaksparenteser) på samme nivå vil bli behandlet i rekkefølge fra venstre mot høyre, og totalt vil derved nodene bli behandlet i *postfiks* rekkefølge.

Som nevnt tidligere kan man se på en nedenfra–opp parsering som at vi fortar suksessive reduksjoner (altså erstatter en grunnfrase med det tilsvarende metasymbolet), inntil vi bare står igjen med startsymbolet. Ser vi nøyere på rekkfølgen som disse reduksjonene foretas i ved en LR-parsering, ser vi at de foretas i omvendt rekkefølge av en *høyre*avledning (og noen mener at R-en i LR stammer fra dette faktum). Dette tilsvarer at hver gang vi under LR-parsering gjør en reduksjon, så er det grunnfrasen lengst til venstre som blir redusert.

LR(k)-grammatikker

På samme måte som for LL-grammatikker, sier vi at en grammatikk er LR(k) dersom man teoretisk sett alltid har informasjon nok til å bestemme alle høyresyntaksparentesene i den delen man *har* lest ved å tillate seg å titte k symboler inn i den delen man ennå ikke har lest (altså se på de k neste klarsymbolene). De aller fleste grammatikker som brukes i praksis (inklusive de eksempler vi har sett på) er LR(1). Det eksisterer en stor teori for hvordan man for en gitt k kan avgjøre om en grammatikk er LR(k), og hvordan man i så fall kan lage en parseringsalgoritme for denne grammatikken.

Allerede for $k = 1$ har imidlertid tabellverket for denne algoritmen en tendens til å bli overveldende stort for vanlig forkommende grammatikker. I praksis holder man seg derfor gjerne til $k = 0$, selv om man da i utgangspunktet bare kan behandle veldig spesielle grammatikker. Ved å titte på klarsymbolet, og å gjøre noen betrakninger over etterfølgermengder, skal vi imidlertid klare å få fram en variant av LR(0)-algoritmen (SLR-algoritmen) som er kraftig nok til å behandle mange praktiske grammatikker, men slett ikke alle. Vi understreker imidlertid at denne SLR-algoritmen langt fra er kraftig nok til å parsere alle grammatikker som teoretisk sett tilfredstiller LR(1)-kravet.

Datastrukturen ved LR-parsering

Når vi utfører en LR-parsering skal vår filosofi være at så fort vi bestemmer en høyresyntaksparentes, så skal vi gjøre den tilsvarende reduksjonen. Hvordan vi klarer å bestemme disse høyreparentesene skal vi komme tilbake til siden. Foreløpig kan vi tenke oss at disse åpenbares oss ad magisk vei, eller vi kan rett og slett tenke oss at de er satt inn på rett plass i den teksten vi skal analysere. Husk at en syntaksparentes også har knyttet til seg et tall som angir hvilken produksjon de tilsvarer.

Under utførelse av LR-parsering vil vi, mer eksplisitt enn ved LL-parsering, lagre den reduserte setningsformen vi til enhver tid har, og den vil være lagret i to deler. Den delen vi *har* lest, og som vi har gjort en del reduksjoner i, kalles (av grunner vi straks skal forklare) *stakken*, mens den delen vi *ikke* har lest ligger helt ubehandlet f.eks. på en fil. Klarsymbolet hører egentlig med til den siste delen, men som før er det hele organisert slik at vi kan titte på dette uten at det regnes for lest. Den setningsformen vi har i øyeblikket, får vi frem ved først å lese stakken fra bunn til topp, og så den uleste delen av setningen, fra venstre mot høyre.

Hovedlinjen i en LR-parsering

Ut fra denne datastrukturen, ser vi at en LR-parsering vil forgå ved at man gjentar følgende operasjoner om og om igjen:

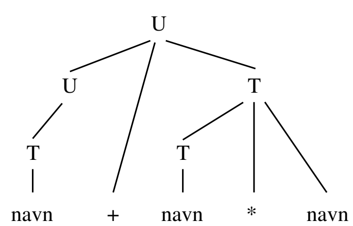
- Les et nytt grunnsymbol, og legg det på toppen av stakken.
- Bestem de eventuelle høyresyntaksparenteser som måtte ligge mellom stakken og klarsymbolet, og utfør de tilsvarende reduksjoner i rekkefølge innenfra og ut (altså i den rekkefølge høyreparentesene står).

Vi kan nå også forstå hvorfor den delen av setningen som er lest kalles en stakk. Det kommer av at det å gjøre en reduksjon tar form av en stakkoperasjon, ved at de symboler som tilsvarer høyresiden i den aktuelle produksjonen fjernes fra toppen av stakken, og at man erstatter dem med det tilsvarende metasymbolet.

Som eksempel kan vi se på en setning i den uttrykksgrammatikken vi tidligere har sett. Vi skriver først opp grammatikken (men med forkortede metasymboler):

$$
\begin{array}{lcll}
\mathrm{U} & \rightarrow & \mathrm{U} + \mathrm{T} & (1) \\
\mathrm{U} & \rightarrow & \mathrm{T} & (2) \\
\mathrm{T} & \rightarrow & \mathrm{T} * \mathbf{navn} & (3) \\
\mathrm{T} & \rightarrow & \mathbf{navn} & (4)
\end{array}
$$

Vi ser på setningen "**navn** + **navn** $*$ **navn**" som har følgende syntaktiske struktur:



Deretter viser vi de enkelte steg i parseringen, ut fra kunnskap om høyresyntaksparentesene (som vi altså rett og slett antar er satt inn i setningen). Parseringen vil da gå slik:

Stakk	Ulest del
-	**navn** $_4)$ $_2)$ + **navn** $_4)$ ∗ **navn** $_3)$ $_1)$
navn	$_4)$ $_2)$ + **navn** $_4)$ ∗ **navn** $_3)$ $_1)$
T	$_2)$ + **navn** $_4)$ ∗ **navn** $_3)$ $_1)$
U	+ **navn** $_4)$ ∗ **navn** $_3)$ $_1)$
U +	**navn** $_4)$ ∗ **navn** $_3)$ $_1)$
U + **navn**	$_4)$ ∗ **navn** $_3)$ $_1)$
U + T	∗ **navn** $_3)$ $_1)$
U + T ∗	**navn** $_3)$ $_1)$
U + T ∗ **navn**	$_3)$ $_1)$
U + T	$_1)$
U	-

Vi skal her legge merke til at dersom vi løpende klarer å bestemme høyresyntaksparentesene, inklusive hvilken produksjon de svarer til, så er det aldri noen tvil om hvor venstreparentesene er. Toppen av stakken vil jo alltid svare til høyresiden av den aktuelle produksjonen, og venstreparentesen vil dermed ligge umiddelbart under (altså foran) disse symbolene.

Vi har i betrakningene over implisitt antatt at vi har å gjøre med en setning som følger språkets regler (og som derfor lar seg parsere på én og bare én måte). Vi skal også i fortsettelsen anta dette, men i spesielle avsnitt lenger ut se på hva som skjer når det er noe galt i setningen. Den entydige BNF-grammatikken som beskriver våre setninger kaller vi G. For å unngå fiklete spesialtilfeller vil vi som nevnt også anta at G ikke har tomme høyresider. Det er imidlertid fullt mulig å gjennomføre det følgende også for grammatikker med tomme høyresider. Dette er kort drøftet i slutten av avsnitt 4.3.4.

4.3.1 Det som kan ligge på stakken utgjør et regulært språk

Det som ligger på stakken under en LR-parsering vil være en blanding av metasymboler og grunnsymboler, og vi skal nå definere et nytt språk R der alfabetet består av både grunn- og metasymbolene i G. Setningene i R skal rett og slett være de symbolsekvenser som noensinne kan ligge på stakken (lest fra bunn til topp) under en LR-parsering av setninger i G.

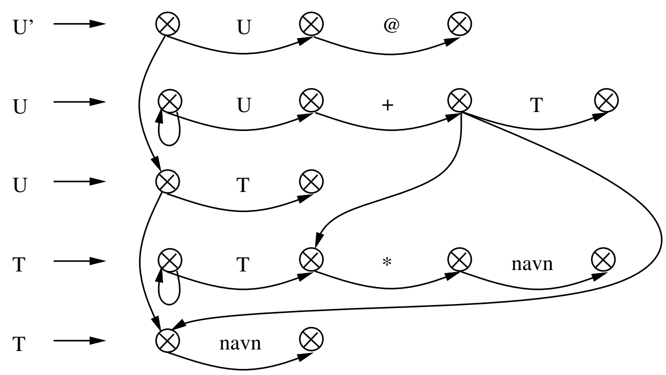
Et sentralt punkt for vår videre teori vil være at *språket R faktisk er regulært*. Det vil vi vise ved å lage en ID-automat som beskriver nettopp dette språket. Vi skal først se på hvordan denne ID-automaten kan konstrueres, og etterpå skissere et bevis for at den framstiller nettopp de strenger som kan ligge på stakken.

For å forenkle de betrakningene vi siden skal gjennom, skal vi også her anta at det helt “ytterst” i grammatikken G er lagt inn en spesiell produksjon: “S’ → S @” (der hverken S’ eller ’@’ er brukt som symboler tidligere). Alle setninger i språket får dermed et entydig sluttmerke ‘@’.

Automaten for det nye språket R konstrueres ved først å skrive opp hver av produksjonene i grammatikken én gang. I hver høyreside legger man så inn en ID-tilstand mellom hvert av symbolene, samt én tilstand helt foran og én helt bakerst (se figuren under). Alle tilstander i automaten skal være sluttilstander. Starttilstanden er den som står helt først i høyresiden i den spesielle ytterste produksjonen.

Kantene, eller tilstandsovergangene, i ID-automaten lages som følger: Fra hver tilstand som ikke ligger på slutten av en høyreside, lages en kant til den neste tilstanden i denne høyresiden. Denne kanten merkes med det (grunn- eller meta-) symbolet som ligger mellom tilstandene. Videre lager vi fra hver tilstand som står foran et metasymbol M, en ϵ-kant til den første tilstanden i hver av produksjonene som har M som venstreside. Legg merke til at det derved også kan bli en kant tilbake til den første tilstanden i den produksjonen man er i, og dermed ofte tilbake til tilstanden selv (dersom produksjonen har venstrerekursjon).

For vår eksempelgrammatikk vil denne automaten se ut som følger. Tilstandene er her tegnet som sirkler (med kryss i, for å markere at de er sluttilstander), og kanter som ikke har noe tegn rett over seg, er ϵ-kanter. Tilatanden øverst til venstre er starttilstand.



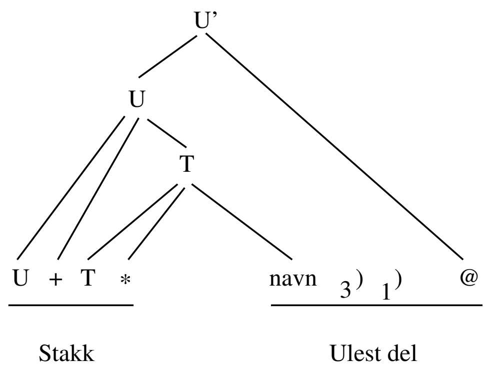
Vi skal siden konvertere denne ID-automaten til en D-automat på den måten vi har sett på tidligere.

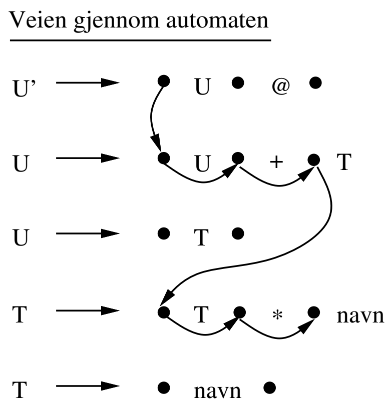
For nå å vise at denne automaten beskriver nøyaktig de strenger som kan ligge på stakken, må vi gjøre to ting:

A Velge en tilfeldig situasjon under LR-parsering av en eller annen setning i språket, og vise at det som ligger på stakken kan framkomme fra automaten over.

B For en vilkårlig setning S_R produsert av automaten R, vise at vi kan konstruere en setning S_G i språket generert av G, og en situasjon under parseringen av S_G der innholdet av stakken er nøyaktig likt S_R.

Før vi gir en skisse av beviset for dette, gir vi et eksempel på hvordan automaten over kan generere innholdet av stakken på et passelig stadium under parseringen av setningen "**navn** + **navn** * **navn**" (se eksempelet på side 35). Vi har også tegnet inn det halvveis reduserte syntakstreet over den setningsformen vi i øyeblikket har, og det vil være forholdet mellom (venstre kant av) dette og den aktuelle veien gjennom automaten (til høyre i figuren) som vil være nøkkelen i beviset.





For å vise punkt A må vi altså tenke oss en generell situsjon i parseringen av en eller annen setning i språket til G. La oss kalle det (grunn- eller meta-) symbolet som ligger på toppen av stakken for “toppsymbolet”. I eksempelet over er toppsymbolet ‘$*$’.

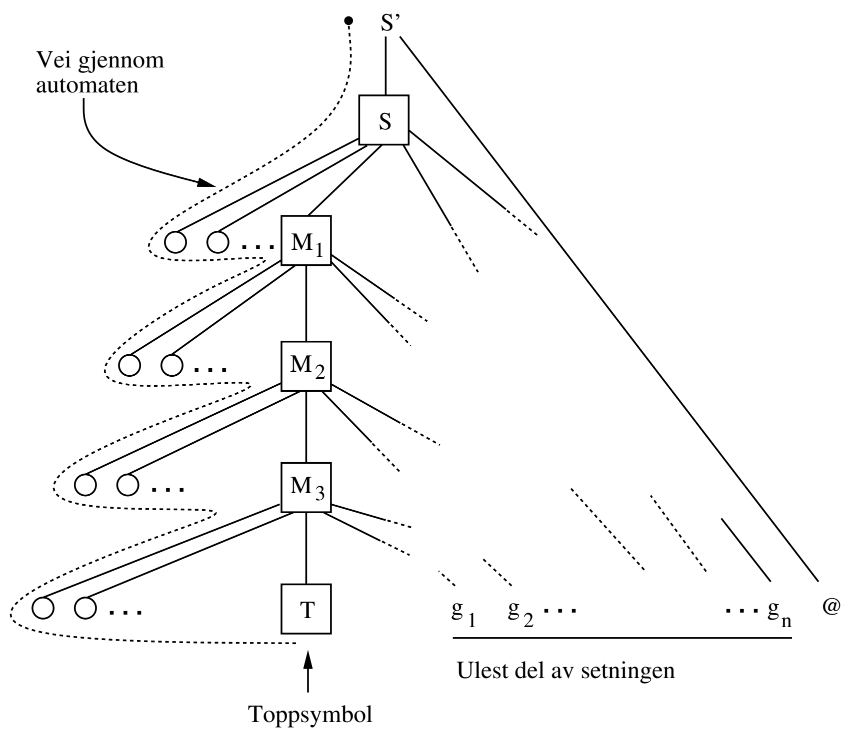
Siden vi her begrenser oss til entydige grammatikker vil det alltid bare finnes ett syntakstre for hver setningsform (delvis redusert setning). Slik dette treet er bygget opp vil vi, ved å gå fra roten og nedover i treet mot toppsymbolet, passere gjennom en sekvens av produksjoner der venstresiden i alle unntatt den første er et bestemt metasymbol i høyresiden i den forrige. Venstresiden i den øverste er selvfølgelig S' (i eksempelet over U').

I hver produksjon i denne sekvensen, unntatt den siste, kan vi altså si at det er et “uthevet” metasymbol i høyresiden, nemlig det som er utviklet videre på neste nivå på vei ned mot topp-symbolet. I den siste produksjonen i sekvensen skal vi si at toppsymbolet er det uthevede symbol (men i motsetning til de andre uthevede, kan dette også være et grunnsymbol, slik som i eksemplet over). For eksempelet over er denne sekvensen følgende, der de uthevede symboler er merket med en firkant.

$$\text{U'} \rightarrow \boxed{\text{U}}\ @ \qquad \text{U} \rightarrow \text{U} + \boxed{\text{T}} \qquad \text{T} \rightarrow \text{T}\ \boxed{*}\ \textbf{navn}$$

I følge LR-algoritmen skissert på side 35, skal vi redusere en høyreside til metasymbolet på venstresiden straks vi kommer til en høyresyntaksparentes. Dette betyr at det aldri er noen høyresyntaksparenteser på stakken. Inne i stakken finnes altså ingen ureduserte fraser, hvilket igjen vil si at til venstre for “linjen av uthevede symboler” i treet vil det ikke finnes noen noder mellom de uthevede symbolene, og symbolene på stakken. Symbolene på stakken er alltid direkte subnoder til en “uthevet node”.

Dermed vil altså det som er på stakken (fra bunn til topp) rett og slett være symbolene foran de uthevede symboler i hver produksjon satt pent etter hverandre, med topp-symbolet satt på til slutt.



Tegningen over viser skisse av det delvis reduserte treet under parseringen. I dette "juletreet" er de uthevede symbolene markert med bokser. Sirklene til venstre for de uthevede symbolene vil være meta- og grunnsymboler. Disse symbolene, regnet ovenfra og ned, vil utgjøre stakken fra bunn til topp, med toppsymbolet helt på toppen.

For å vise punkt A må vi påvise at det finnes en vei gjennom den ID-automaten vi beskrev tidligere, som er slik at den genererer nettopp det som ligger nå på stakken. En slik vei kan vi finne som følger: Fra starttilstanden (først i høyresiden til S'-produksjonen) går vi ved en ϵ-kant til starten av den S-produksjonen som er brukt. Vi fortsetter så utover denne til tilstanden foran det uthevede symbol. Fra denne tar vi igjen en ϵ-kant til starten av den produksjonen som så er brukt, og fortsetter på samme måten. Merk at vi på veien kan være flere ganger innom samme produksjon, men gjerne med forskjellige uthevede symboler.

Dette tilsvarer at vi følger venstre kant av "juletreet" i figuren over, slik den stiplede linjen viser. Dette vil til slutt bringe oss fram til toppsymbolet, og da vil vi ha generert nøyaktig den symbolsekvensen som ligger på stakken.

Dermed har vi skissert et bevis for del A. Det å vise del B koker ned til å gjøre den samme konstruksjonen i omvendt rekkefølge. Vi må da tenke oss at vi har en setning i språket R, som altså er generert av en vei gjennom ID-automaten. Ut fra denne veien må vi så konstruere et halvredusert syntakstre av samme type som over, og som ville gi den riktige veien ved konstruksjonen gjort for del A. Dette skulle ikke by på prinsippielle problemer og er overlatt den iherdige leser.

4.3.2 Transformasjon til deterministisk automat

Vi skal nå transformere den ID-automaten som beskriver de mulige stakker, til en D-automat, slik vi har vært gjennom i kapittelet om regulære språk. Tilstandene i D-automaten blir da altså mengder av tilstander fra ID-automaten. Vi trenger derfor en måte til greit å kunne angi slike ID-tilstander. Hver ID-tilstand er gitt ut fra dens plassering i høyresiden av en bestemt produksjon. En grei og informativ notasjon for dette er å skrive opp hele produksjonen med en prikk som angir tilstandens plassering. En slik "produksjon med prikk" skal vi kalle et *LR-element* (engelsk: LR-item), eller bare et *element*.

Fra produksjonen "T → T ∗ **navn**" får vi altså de fire elementene

$$\mathrm{T} \rightarrow \textbf{.}\ \mathrm{T} * \mathrm{navn} \qquad \mathrm{T} \rightarrow \mathrm{T}\ \textbf{.} * \textbf{navn} \qquad \mathrm{T} \rightarrow \mathrm{T} *\ \textbf{. navn} \qquad \mathrm{T} \rightarrow \mathrm{T} * \textbf{navn .}$$

Hvert element svarer altså nøyaktig til en ID-tilstand, og i det følgende vil vi ikke skille mellom disse termene, men bruke dem om hverandre Elementnotasjonen er nyttig til mer enn å konvertere ID-automaten til en D-automat. Så vi tar med to definisjoner som vi trenger senere:

Start-element og indre element: Et *start-element* er et LR-element der prikken står først i høyre-siden. De øvrige elementer, også de der prikken står helt bakerst, kalles *indre elementer.*

Merk at vi antar at alle produksjoner har minst ett symbol i høyresiden, og det er derfor ingen tvil om hva som er start-elementer og indre elementer.

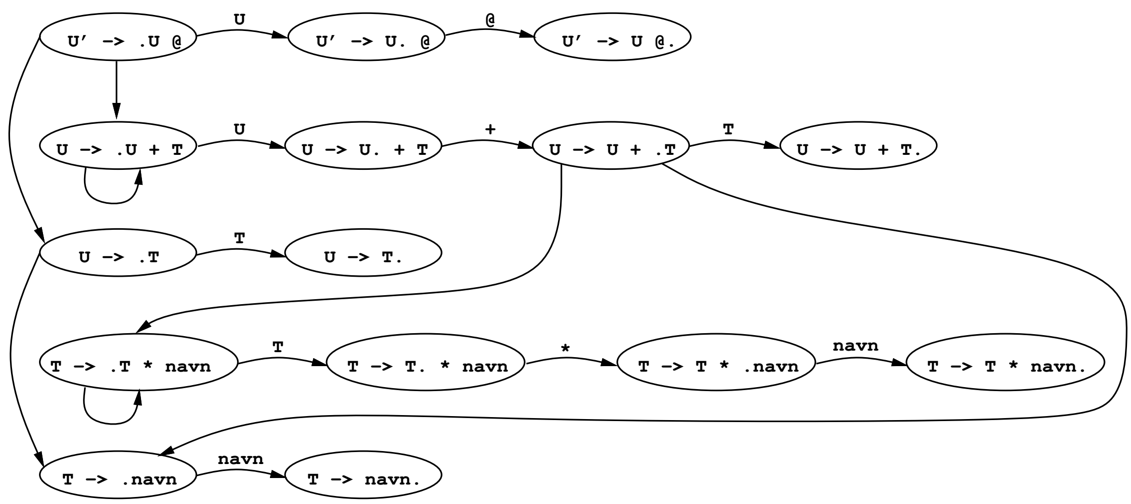
Når man har gjort en del steg i LR-parseringen av en (riktig) setning vil man ha en situasjon som antydet ved juletre-figuren over, der trestrukturen er entydig gitt. Merk imidlertid at treet bare er entydig gitt dersom også den *uleste* delen er gitt. Om vi bare kjenner det som er på stakken kan det være flere mulige trær, men disse vil da tilsvare forskjellige fortsettelser. For å kunne snakke om situsjonen omkring toppen av stakken, gjør vi følgende definisjon:

Det aktive element : Anta at vi er på et tilfeldig trinn i LR-parseringen av en (riktig) setning S, og at stakken ikke er tom. Vi definerer da *situasjonens aktive element* til å være det element som dannes fra den produksjon i syntakstreet der toppsymbolet inngår i høyresiden, og som har prikken satt etter toppsymbolet.

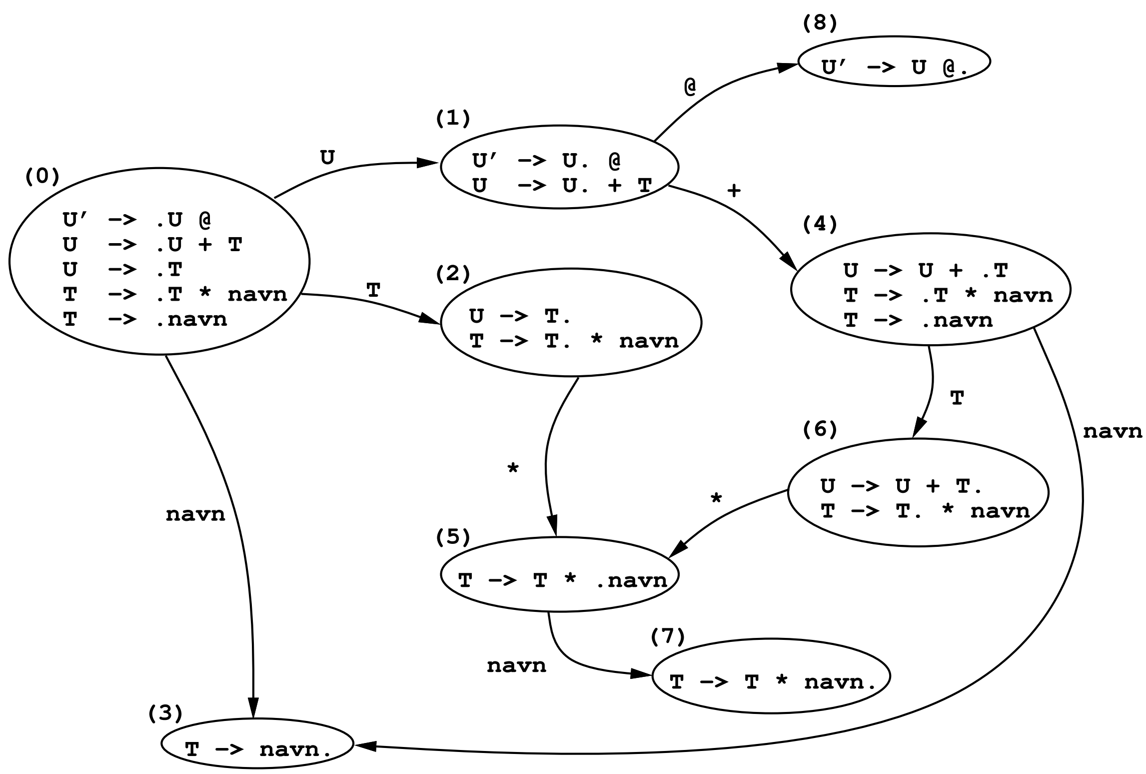
I juletre-figuren er det aktive element "$\mathrm{M}_3 \rightarrow \cdots \mathrm{T}\ \textbf{.} \cdots$". Det aktive element er alltid et indre element, siden prikken står bak toppsymbolet. Det aktive element i situasjonen angitt i figuren på side 38 er "T → T ∗ **. navn**".

Merk altså at det aktive element ikke er definert dersom stakken er tom, men det er lett å se at det bare bare er tilfelle helt i starten av en LR-parsering. Siden vi ikke har tomme høyresider i produksjonene er nemlig alltid det første vi skal gjøre i en LR-parsering å lese første grunnsymbol i S over på stakken.

Med elementnotasjonen kan den ID-automaten som tilsvarte vår grammatikk tegnes slik:



Når vi nå skal gjøre om denne ID-automaten til en D-automat, må vi følge de generelle reglene for tillukking etc. som ble angitt for dette i kapittel 3. Det er da ikke vanskelig å se at den deterministiske automaten vi får fra eksempelet over, blir som følger:



Merk her at også i denne automaten vil alle tilstander være sluttilstander (uten at vi har avmerket dette på noen måte). Når man skal sette opp denne D-automaten er det ikke noe problem å gå direkte fra grammatikken, til den deterministiske automaten. Vi må da oversette stegene i denne konstruksjonen til terminologien med LR-elementer, og vi får da følgende oppskrift:

- Først definerer vi tillukningen av en mengde L av LR-elementer.

  Steg: Dersom det i L er med et element med prikk foran et metasymbol M, så inkluderer vi i L alle elementer på formen "M → . ⋯"

  Gjenta steget inntil elementmengden ikke øker mer.

- Tilstandene i automaten vil være mengder av LR-elementer. Konstruksjon av tilstander og kanter i D-automaten foregår som følger:

  Initialisering: Starttilstanden består av tillukningen av mengden { S'→ . S @ }.

  Steg: Se på en allerede definert tilstand T, og la x være et meta- eller grunnsymbol. Det skal gå en x-kant fra T dersom det i T finnes elementer av formen "M → ⋯ .x⋯". Denne kanten skal gå til en tilstand som inneholder alle de tilsvarende elementer "M → ⋯ x. ⋯", samt tillukningen av disse. Hvis en slik tilstand ikke allerede finnes legges denne inn som en ny tilstand.

  Gjenta steget inntil det ikke er mulig å lage flere kanter.

4.3.3 Mot en parseringsalgoritme

Vi konstaterte lenger opp at en LR-parsering skal gjøres ved å gjenta en enkel operasjon inntil setningen er lest og alle reduksjoner er gjort. Litt omskrevet kan denne operasjonen skrives:

LR-steget:

Sjekk først om det skal være noen høyre-syntaksparenteser mellom topp-symbolet og klar-symbolet, og på grunnlag av det, utfør følgende:

- *Dersom det skal være høyreparentes, så utfør den reduksjon som den første (innerste) av disse parentesene indikerer.*
- *Ellers: les neste symbol, og legg det på toppen av stakken.*

Inntil nå har vi antatt at de syntaktiske høyreparentesene er åpenbart oss ad magisk vei. Men vi kan ikke alltid regne med å være så heldige, så vi er ute etter en mer håndfast metode for å avgjøre dette, og det vi sitter med av informasjon er det som ligger på stakken, samt eventuelt klarsymbolet.

Det man hele tiden må ta stilling til er altså om det i den aktuelle situsjon er riktig å gjøre en reduksjon, eller å lese inn neste symbol. I det første tilfellet må vi også finne ut hvilken reduksjon som er den riktige. Vi skal kalle disse operasjonene h.h.v. for et *reduksjonssteg* og et *lesesteg*. (Merk at et lese-steg i engelsk litteratur gjerne kalles et "shift"-steg. LR-parsering kalles derfor ofte "shift-reduce-parsing").

Tolking av D-automaten

Når vi har en LR-parsering gående, vil det som ligger på stakken hele tiden være en setning i R. Det kan derfor leses og godkjennes av den D-automaten vi har konstruert fra grammatikken, slik som vist over. La oss så tenke oss at en LR-parsering av en setning i G har pågått en stund, og at vi på et visst tidspunkt får vite hva som ligger på stakken (men ikke hvilke reduksjoner som har vært gjort, og ikke hva den uleste delen av setningen er).

Om vi kjenner G kan vi nå lage den tilsvarende D-automaten, og vi kan kjøre stakken gjennom denne. La oss så se på den tilstanden D-automaten er i etter å ha lest hele stakken. Denne tilstanden skal vi kalle *topptilstanden*, og vi vil forsøke å finne ut hva denne sier oss om hvordan situasjonen er omkring det stedet vi nå er i setningen. Det vi selvfølgelig er interessert i er å finne ut *om* det er en høyresyntaksparentes mellom toppsymbolet på stakken og klarsymbolet, og i så fall *hvilken* produksjon den tilsvarer. Det følgende lemmaet er avgjørende i denne forbindelse.

Det sentrale LR-lemma:

Anta at vi utfører LR-parsering av en setning produsert av en grammatikk G, og at vi på et visst tidspunkt der stakken ikke er tom, lar det som er på stakken gå gjennom den D-automaten som framkommer fra G ved konstruksjonen over, og at denne ender opp i (topp-)tilstanden DT. Om vi nå tenker oss alle syntaktisk korrekte fortsettelser av setningen (og av disse er det gjerne uendelig mange) og for hver av disse ser på situasjonens aktive element, så vil disse utgjøre nøyaktig de indre elementer i DT

Sentralt i beviset av dette lemmaet står *Endlig-automat-lemmaet* som vi så på i forbindelse med regulære språk. Vi gjengir derfor først dette.

Endelig-automat-lemmaet:

Anta at en D-automat er konstruert fra en ID-automat etter reglene med tillukning etc., og at denne D-automaten har lest en viss sekvens av grunnsymboler, og dermed kommet i en viss tilstand DT. Da vil de ID-tilstander som DT er bygget opp av, angi nøyaktig de ID-tilstander som ID-automaten kunne være i etter å ha lest (eller om man vil: produsert) denne sekvensen av grunnsymboler.

La oss så gå tilbake til LR-lemmaet, og la DT være den tilstanden som D-automaten er i etter å ha lest hele stakken. Beviset av det sentrale LR-lemma er nå egentlig bare å sammenholde den kunnskap vi nå sitter med på en rett etter nesa måte. Å skrive det ut i detalj ville imidlertid kreve en del plass, så vi gir bare en skisse og oppfordrer leseren til gå gjennom logikken på egen hånd. Beviset kan passelig deles i følgende to deler:

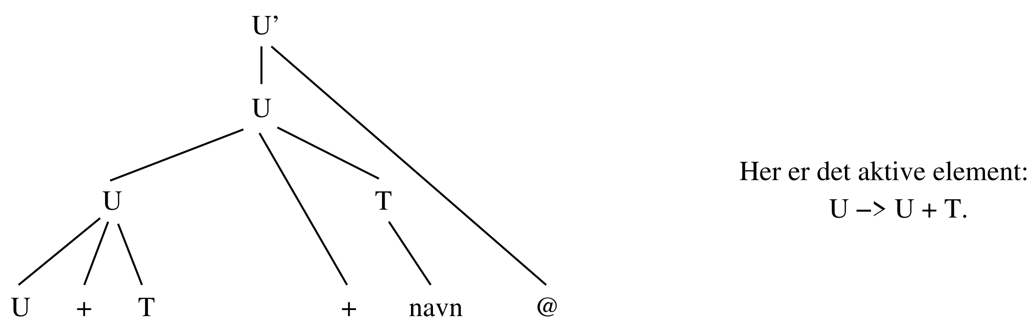
Vi viser først at for hver syntaktisk korrekt fortsettelse av stakken, så vil situasjonens aktive element være å finne som et indre element i DT. Vi tenker oss da en riktig fortsettelse, og lager det entydig bestemte syntakstreet over den setningsformen som dannes av stakken og fortsettelsen. Denne situasjonen vil da ha et aktivt elemnt, og dette er hvertfall helt sikkert et indre element.

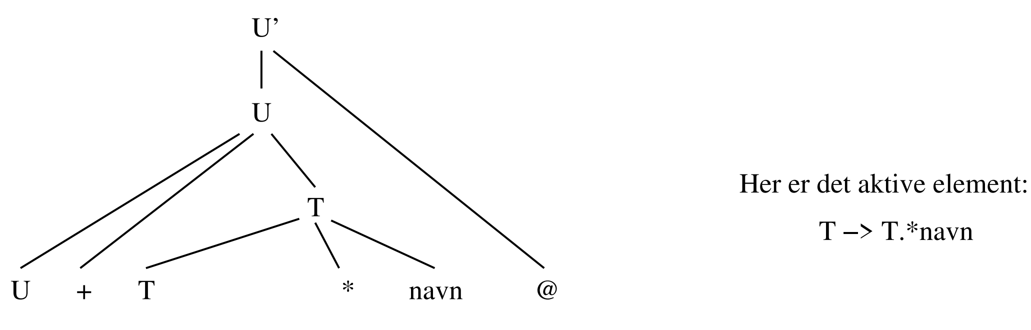
Ut fra dette syntakstreet velger vi så å gå gjennom ID-automaten på den måten som angis av veien gjennom dets uthevede symboler (se den stiplede linjen i juletre-figuren på side 39). Det siste uthevede symbolet er toppsymbolet og etter å ha brukt dette i ID-automaten er vi i den ID-tilstand (= element) som nettopp er situasjonens aktive element. Siden denne måten å gå gjennom ID-automaten på vil produsere nøyaktig det som er på stakken, ser vi av *Endelig-automat-lemmaet* at dette elementet også må være med i topp-tilstanden DT. Dermed er LR-lemmaet vist den ene veien.

For å vise det den andre veien, må vi for hvert indre element E i DT vise at det finnes en fortsettelse av det som er på stakken, slik at situasjonens aktive element nettopp er E. For å få tak i en slik fortsettelse kan vi foreta konstruksjonen over i omvendt rekkefølge. Man starter da med den ID-tilstanden som tilsvarer E. Ut fra *Endelig-automat-lemmaet* kan vi så finne en vei gjennom ID-automaten som produserer den gitte stakken, og som samtidig ender i denne tilstanden. Deretter lager vi et syntaks-tre som har en sekvens av uthevede symboler som tilsvarer denne veien. Til slutt lar vi dette treet produsere en passelig fortsettelse ved å lage en eller annen avledning til høyre for de uthevede symbolene. Detaljene her skulle være kurante, og dette avslutter dermed beviset for det sentrale LR-lemmaet.

Det sentrale LR-lemmaet forteller oss altså at om vi har gitt en ikketom stakk, så er det kun de indre elementene i den D-tilstanden som stakken bringer oss til (topptilstanden) som kan være aktuelle kandidater for situasjonens aktive element. Ved å se gjennom disse indre elementene få vi derved et godt inntrykk av hva slags reduksjon eller lesing som kan være aktuell i den foreliggende situasjon.

For å illustrere dette lemmaet kan vi vende tilbake til den uttrykksgrammatikken vi har sett på tidligere, og den tilhørende D-automaten. La oss tenke oss at "U + T" ligger på stakken. Vi ser da at D-automaten vil gjøre tilstand (6) til topptilstand. Inne i denne tilstanden finner vi to elementer (som begge er indre elementer). I følge lemmaet skulle det da gå an å konstruere fortsettelser som gjør at nettopp disse elementer blir de aktive elementer. To slike fortsettelser er vist under:





Eksempelet over illustrerer en annen viktig egenskap som følger av det sentrale LR-lemmaet. For hvert av de indre elementene i topptilstanden vil den symbolsekvensen som står foran prikken (i høyresiden) være den samme som ligger på toppen av stakken (med det som er nærmest prikken øverst). Legg også merke til at dette betyr at det som står foran prikken i alle indre elementer er likt – regnet fra prikken og så langt det rekker. Denne egenskapen gjør at vi ikke trenger å tenke på hvorvidt de reduksjoner vi vil foreslå senere er mulig å gjennomføre. Det vil de alltid være.

Betydningen av start-elemenetene

Det er nå også mulig å få tak i betydningen av de *start-elementene* vi finner i en D-tilstand, som for eksempel i tilstand 4 i automaten på side 41. Ut fra hvordan denne automaten konstrueres direkte fra grammatikken, ser vi at dette er direkte eller indirekte avledninger av metasymboler som står bak prikken i et mulig aktivt element. Disse sier derfor, ut fra en gitt stakk, nøyaktig hvilke venstresyntaksparenteser som kan ligge mellom toppsymbolet og klarsymbolet (men ikke nødvendigvis som første slik parentes). Leseren oppfordres til å lage tilsvarende figurer som den over, som viser dette. Stakken kan da f.eks. inneholde: "U +".

Start-tilstanden i D-automaten er litt spesiell, i og med at den alltid bare vil inneholde start-elementer. Ut fra betrakningen over ser vi at disse start-elementene representerer nøyaktig de venstre-syntaksparenteser som kan kan stå foran første symbol i en eller annen riktig setning i språket.

Start-elementene angir altså mulige venstre-syntaksparenteser, men siden LR-parsering dreier seg om å finne høyre-syntaksparentesene er ikke start-elementene så viktige i denne sammenheng.

4.3.4 LR(0)-algoritmen

Under en LR-parsering må man hele tiden ta stilling til om det i den aktuelle situasjonen er riktig å lese inn et nytt symbol til stakken, eller å redusere noe som ligger på toppen av stakken. Vi skal nå se at dersom vi er riktig heldige med hvordan elementene fordeler seg i D-tilstandene, så kan vi faktisk ta denne avgjørelsen ved bare å se på elementene i den D-tilstanden som ligger på toppen av stakken.

Vi skal imidlertid først definere en ny inndeling av LR-elementene, og denne gangen bygger inndelingen på om prikken står bakerst eller ikke. Vi skal snakke om *reduksjonselementer* og *leseelementer*.

Dersom vi i topptilstanden har et element med en prikk til slutt, vil dette si at det finnes en mulig fortsettelse av setningen, som gjør at situasjonens aktive element har prikken helt på slutten av høyresiden. Vi er da helt på slutten av "den aktive produksjon", og det betyr at det riktige nå vil være å gjøre den tilsvarende reduksjonen. Vi kaller derfor et slikt element for et *reduksjonselement*.

Et element som ikke har prikken på slutten indikerer at vi ikke er ferdige med den tilsvarende frasen og at, om dette er det aktuelle element, så skal vi gjøre en leseoperasjon. Vi kaller derfor et slikt element for et *lese-element*.

Vi kan dermed karakterisere en type grammatikker som er spesielt greie å ha med å gjøre. Karakteriseringen går på hvordan LR-elementene fordeler seg i tilstandene til deres D-automat

LR(0)-kravet

En grammatikk sies å oppfylle LR(0)-kravet dersom alle tilstandene i dens D-automat oppfyller ett av følgende to krav:

- D-tilstanden inneholder ett og bare ett element, og dette er et reduksjonselement. Vi kaller en slik D-tilstand en *reduksjonstilstand*. Den aksjonen som skal gjøres dersom en slik reduksjonstilstand ligger på toppen av stakken, er å gjøre den tilsvarende reduksjonen.
- Alle elementer i D-tilstanden er leseelementer. Vi skal kalle en slik D-tilstand en *lesetilstand*. Dersom en lesetilstand ligger på toppen, skal det gjøres et lesesteg.

Dersom en grammatikk oppfyller dette LR(0)-kravet ser vi at den topptilstanden som den aktuelle stakken angir alene vil avgjøre hviket steg som er riktig å gjøre.

Vi skal her knytte et par kommentarer til forholdet mellom

Start-tilstander er naturlig å se som en lese-tilstand, men de tilsvarer ikke noe aktivt element. ... Lett å se følgende: Start-tilstanden i D-automaten vil inneholde bare startelementer, og siden Start-tilstanden i D-automaten vil bli topptilstand bare når stakken er tom, og dette inntreffer bare helt i starten

....

For at det skal være mulig å avgjøre om vi skal lese eller redusere bare ved å se på D-tilstanden etter at stakken er lest, må hver av tilstandene i D-automaten oppfylle ett av kravene under:

LR(0)-grammatikker er, som nevnt tidligere, de grammatikkene der man uten en gang å titte på klarsymbolet alltid skal kunne avgjøre hvilket LR-steg som er det riktige. Vi ser at om en grammatikk er slik at den resulterende D-automat tilfredstiller kravet over, så vil tilstanden på toppen av stakken alltid entydig bestemme det neste LR-steget. Den vil altså være en LR(0)-grammatikk.

Det omvendte gjelder også, altså om vi har en LR(0)-grammatikk vil den tilsvarende D-automaten alltid tilfredstille kravet over, men vi skal ikke gå nærmere inn på begrunnelsen for dette. Ut fra dette kalles kravet over LR(0)-kravet.

Konstruksjon av en LR(0)-algoritme

Dessverre er det slik at de fleste av de grammatikker man treffer på i praksis, ikke tilfredstiller LR(0)-kravet. Dette gjelder, som vi ser, også vår eksempelgrammatikk, der tilstandene (2) og (6) ikke tilfredstiller dette kravet. Men dersom vi er så heldig at vår grammatikk tilfredstiller LR(0)-kravet, kan vi nå lett lage en parserings-algoritme.

For en gitt LR(0)-grammatikk lager vi da først den tilsvarende D-automaten. For hver tilstand må vi også markere om den er en lesetilstand eller en reduksjonstilstand. I det siste tilfellet må det også angis hvilken reduksjon som skal gjøres.

Algoritmens steg blir da rett og slett først å la stakken gå gjennom D-automaten, for å finne topptilstanden. Ut fra denne kan man så avgjøre hva den riktige aksjon er.

Når man starter en parsering vil det første steget alltid være et lese-steg. Dette stikker i at ingen produksjoner har tom høyreside, og det stemmer også med at start-tilstanden i D-automaten bare har start-elementer og dermed ingen reduksjons-elementer.

Under en slik parsering kan man selvfølgelig la *hele* stakken gå gjennom automaten hver gang det har foregått en forandring, for å finne topptilstanden. Man ser imidlertid lett at dette kan effektiviseres, siden man i hvert steg bare forandrer stakken litt i toppen. Man kan i stedet utvide stakken slik at man mellom hvert symbol (samt under bunnsymbolet og over toppsymbolet) lagrer den tilstanden som D-automaten vil være i på veien fra bunnen og opp. Ved en leseoperasjon finner vi da lett den nye topptilstanden ved å følge D-automaten ett hakk. Ved en reduksjon skal man først fjerne en del symboler på toppen (tilsvarende høyresiden), og da må man også fjerne de tilstander som ligger mellom og over disse. Det nye metasymbolet legges så på, og igjen blir det bare én tilstands-overgang for å finne den nye topptilstanden.

Feilfinning

Programmerere gjør ofte syntaktiske feil, og vi ønsker at syntaksanalysatoren skal si fra så fort den har lest et symbol som gjør at den leste delen umulig kan være begynnelsen av en riktig setning (program). Ved LR(0)-parsering ut fra algoritmen over viser det seg at dette er greit å få til, og den testen vi da hele tiden må gjøre er at det vi har på stakken er noe som kan godkjennes av D-automaten. Denne D-automaten vil jo nettopp godkjenne *nøyaktig de stakker som kan forlenges til riktige setningsformer ved en passelig fortsettelse*, hvilket betyr at det stadig er håp om at dette skal bli et syntaktisk riktig program.

Det kritiske punktet i algoritmen blir dermed lese-operasjonen, altså den som leser neste symbol og legger det på toppen av stakken. Her får vi en naturlig test ved at vi kan sjekke at den nye stakken også godkjennes av D-automaten. Om vi har tilstander mellom stakk-symbolene, gjøres dette ved å se om det fra den gamle topptilstanden går en kant merket med det innleste symbol. Om det finnes en slik er alt i orden, ellers vil det symbolet vi leste inn ikke kunne aksepteres på dette stedet, og en feil skal signaliseres.

Vi kunne også vurdere å gjøre en tilsvarende test når vi gjør reduksjoner (og etter fjerning av noen symboler, legger et nytt metasymbol på toppen av stakken), men det viser seg at denne testen aldri ville slå ut. Det er nemlig lett å vise (overlates til leseren) at om vi har en stakk som aksepteres av D-automaten, og om vi gjør en reduksjon som indikeres av et element som ligger i topp-tilstanden, så vil også den resulterende stakken aksepteres av D-automaten.

Avslutning av parseringen

Om vi kommer i en situasjon der topptilstanden inneholder elementet "S' $\rightarrow$ S @ .", så vet vi at toppen av stakken vil inneholde symbolene "S @". Akkurat i dette tilfellet vil vi imidlertid vite mer, da vi (ut fra *Endelig-automat-lemmaet*) lett kan vise at det *bare* disse to symbolene som ligger på stakken. Dette vil jo igjen si at den setningen vi til nå har lest, etter alle kunstens regler er redusert til start-symbolet, og at vi til slutt har funnet symbolet '@', som altså betyr at input-strengen er slutt. Vi kan da med god samvittighet signallere at vi har lest en riktig setning i språket.

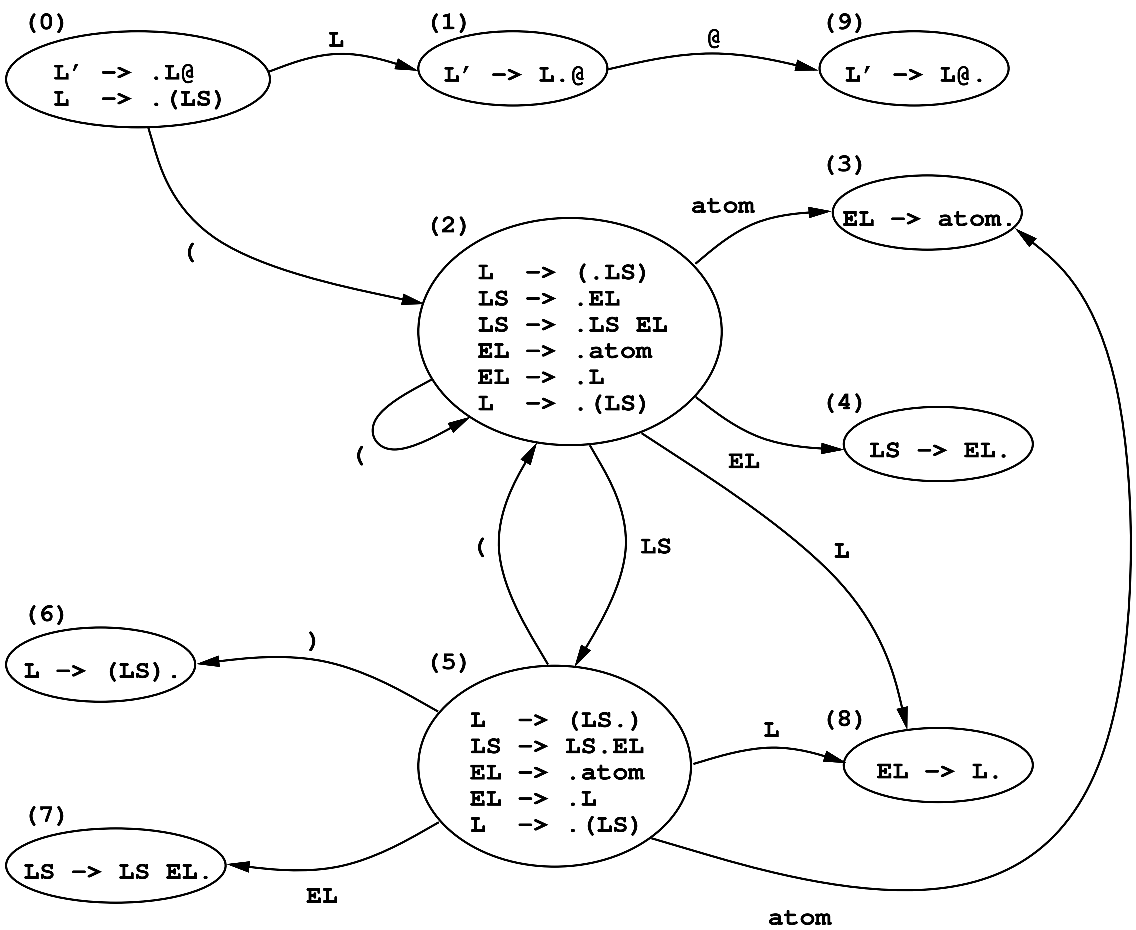
Det er heller ikke vanskelig å innse at om vi gjør riktig LR-analyse av en riktig setning, så vil vi ved slutten av setningen komme i denne topptilstanden. Med dette skulle vi ha en fullstendig algoritme for parsering og sjekking av setninger i LR(0)-grammatikker.

Et LR(0)-eksempel

Vi avslutter dette avsnittet om LR(0)-parsering med et eksempel der vi setter opp D-automaten for en grammatikk, viser at denne er LR(0) og skisser en parserings-algoritme ut fra dette. Som eksempelgrammatikk tar vi følgende Lisp-aktige grammatikk:

L' $\rightarrow$ L @
L $\rightarrow$ (LS)
LS $\rightarrow$ EL | LS EL
EL $\rightarrow$ **atom** | L

Ved å følge oppskriften gitt tidligere i dette kapitlet får vi følgende D-automat:



Vi ser at tilstandene 3, 4, 6, 7 og 8 er reduksjonstilstander, mens de andre er lesetilstander. D-automaten tilfredstiller LR(0)-kravet og grammatikken er en LR(0)-grammatikk. En LR-algoritme kan derfor skisseres som følger:

```
Legg tilstand 0 på stakken;
WHILE topptilstand NE 9 DO
 BEGIN
    IF topptilstand = (0,1,2,5) THEN
     BEGIN
        les et symbol, og legg det på toppen av stakken;
        forsøk å finne ny topptilstand ut fra D-automaten;
        IF  en slik ny tilstand finnes THEN  legg den på toppen av stakken
        ELSE det er feil i input
     END
    ELSE BEGIN
        IF topptilstand = 3 THEN reduser med EL->atom     ELSE
        .                 4                 LS->EL       ELSE
        .                 6                 L ->(LS)     ELSE
        .                 7                 LS->LS EL    ELSE
        .                 8                 EL->L        ;

        finn ny topptilstand ut fra D-automaten (vil alltid gå bra)
     END
 END;
```

Parseringen av setningen "`( a ( a a ) ) @`" (a for atom) vil gå som følger:

```
         Stakk                                 Det uleste
----------------------------             ---------------
0                                        ( a ( a a ) ) @
0 ( 2                                      a ( a a ) ) @
0 ( 2 a 3                                    ( a a ) ) @
0 ( 2 EL 4                                   ( a a ) ) @
0 ( 2 LS 5                                   ( a a ) ) @
0 ( 2 LS 5 ( 2                                 a a ) ) @
0 ( 2 LS 5 ( 2 a 3                               a ) ) @
0 ( 2 LS 5 ( 2 EL 4                              a ) ) @
0 ( 2 LS 5 ( 2 LS 5                              a ) ) @
0 ( 2 LS 5 ( 2 LS 5 a 3                            ) ) @
0 ( 2 LS 5 ( 2 LS 5 EL 7                           ) ) @
0 ( 2 LS 5 ( 2 LS 5                                ) ) @
0 ( 2 LS 5 ( 2 LS 5 ) 6                              ) @
0 ( 2 LS 5 L 8                                       ) @
0 ( 2 LS 5 EL 7                                      ) @
0 ( 2 LS 5 ) 6                                         @
0 L 1                                                  @
0 L 1 @ 9    (Hurra!)                                  -
```

La oss så prøve å parsere en syntaktisk gal setning:

```
         Stakk                               Det uleste
----------------------------             ------------
0                                        ( a ( ) ) @
0 ( 2                                      a ( ) ) @
0 ( 2 a 3                                    ( ) ) @
0 ( 2 EL 4                                   ( ) ) @
```

```
0 ( 2 LS 5                          ( ) ) @
0 ( 2 LS 5 ( 2                        ) ) @
0 ( 2 LS 5 ( 2 ) ?                      ) @
```

Fra tilstand 2 går det ingen kant merket ')', altså feil.

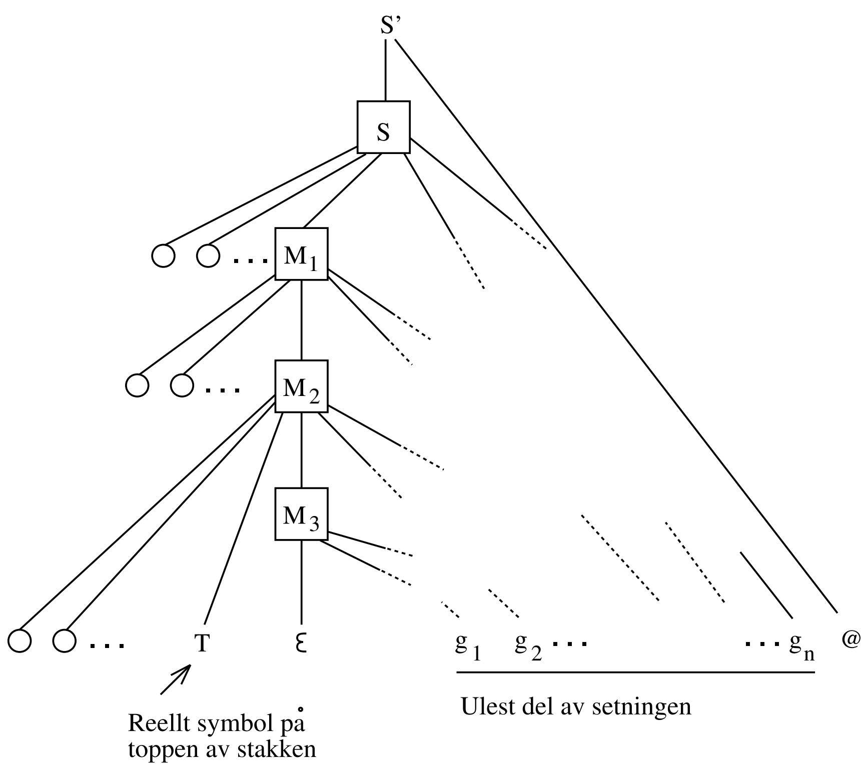
Litt om grammatikker med tomme høyresider

Produksjoner der høyresiden er tom er nyttige i mange forbindelser. De gjør det blant annet lettere å skrive språkdefinisjoner. Et eksempel på en slik produksjon er definisjonen av en klassedeklarasjon i SIMULA:

CLASS_DECLARATION	$\rightarrow$	PREFIX MAIN_PART
PREFIX	$\rightarrow$	ϵ | class_identifier

Det er riktignok slik at tomme høyresider ikke er like tvingende nødvendige i forbindelse med LR-grammatikker som ved LL-grammatikker. I det siste tilfellet nærmest tvinger de tomme høyresidene seg frem når en grammatikk skal skrives om til LL-form. I LR-grammatikker brukes tomme høyresider mer av bekvemmelighetshensyn.

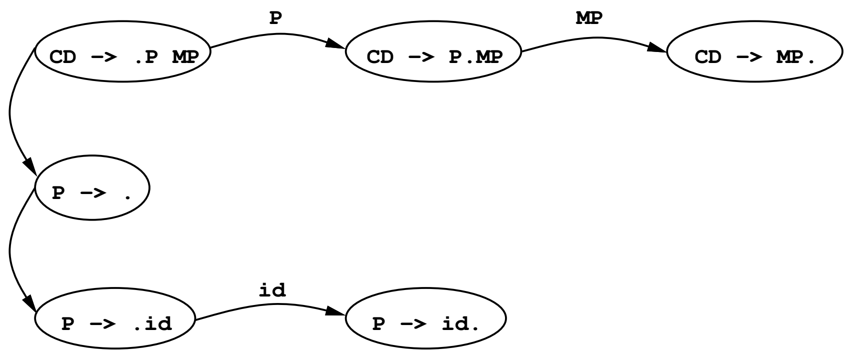
Som tidligere nevnt gjelder mange av de resonnementene vi har skissert over også for entydige grammatikker med tomme høyresider, selv om vi for enkelhets skyld har antatt at slike ikke finnes. Vi skal her se litt på hvordan situasjonen vil forandre seg dersom vi tillater tomme høyresider, og som et utgangspunkt kan vi se på en "juletre-figur" der det er brukt en tom produksjon mellom stakken og klarsymbolet.



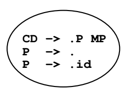
Det er rimelig her å si at produksjoner med en tom høyreside gir opphav til nøyaktig ett LR-element, som *bare* har en prikk på høyresiden. Problemet i situasjonen over blir først og fremst at et slikt element aldri kan være *Det aktive element* ut fra den definisjonen vi har gitt tidligere, men likevel kan det være dette som angir den riktige aksjon som nå skal gjøres. I situasjonen over er det f.eks. nettopp elementet "$M_3 \rightarrow .$" som angir riktig aksjon, nemlig en reduksjon med denne produksjon (som vil føre til at M_3 legges på toppen av stakken, uten at noe leses). Et slikt element må altså rubriseres som et reduksjonselement.

Med denne definisjonen går imidlertid hovedlinjene i teorien gjennom som før, uten at vi skal gå nærmere inn på dette. Det faller ganske naturlig ut hvordan de nye elementene skal inngå når D-automaten settes opp, og ved å regne dem som reduksjonselementer vil også f.eks. LR(0)-kravet og veien fram til en LR(0)-algoritme kunne brukes som før.

Som et eksempel kan vi se på det utsnittet av SIMULA-syntaksen som vi så på over. De tre produksjonene (nå med forkortede symbolnavn) vil gi opphav til følgende ID-tilstander:



Konverterer vi utsnittet av ID-automaten over til en D-automat, får vi blant annet tilstanden vist under.



Som vi ser består denne tilstanden både av et reduksjonselement "$P \rightarrow .$" og en lesetilstand "$P \rightarrow .$ **id**". Dette strider mot LR(0)-kravet, og dette utsnittet av SIMULA-grammatikken er således ikke en LR(0)-konstruksjon.

4.3.5 Grammatikker som ikke er LR(0), og SLR-grammatikker

Som nevnt tidligere er det få praktiske grammatikker som er LR(0)-grammatikker. Slike grammatikker har nemlig en tendens til å bli nokså klumpete, da alle konstruksjoner må avsluttes med en eksplisitt slutt-parentes som inngår i konstruksjonen. Lisp-syntaksen har denne egenskapen, og kan derfor lett formuleres som en LR(0)-grammatikk. Å forlange det tilsvarende f.eks. for alle uttrykk i SIMULA ville derimot føre til en overdreven mengde syntaktisk sukker. Et uttrykk avsluttes i SIMULA ved at det kommer et symbol som ikke hører med i uttrykk (f.eks. semikolon eller "then"), ikke ved å angi et eksplisitt sluttmerke.

Men hvordan skal man da nyttiggjøre seg den LR(0)-teorien vi nå har sett på for andre grammatikker? Det viser seg å være flere måter å gjøre dette, og vi skal her se på den aller enkleste metoden (SLR-grammatikker), og så skissere et par måter å gå videre på

(LALR-grammatikker og generelle LR(1)-grammatikker). Vi understreker igjen at den SLR-algoritmen vi skal beskrive, ikke er kraftig nok til å behandle alle LR(1)-grammatikker.

Nøkkelen til å komme videre er å tillate seg å se på klarsymbolet, for derved kanskje lettere å kunne avgjøre om man med en viss topptilstand skal gjøre en leseoperasjon eller en reduksjon (og i tilfelle hvilken). La oss derfor anta at vår grammatikk ikke tilfredstiller LR(0)-kravet. Dette kravet kan brytes på to måter: enten kan en D-tilstand inneholde flere reduksjons-elementer, eller den kan inneholde både lese- og reduksjons-elementer. I vårt eksempel med grammatikk for aritmetriske uttrykk, brytes LR(0)-kravet på den siste måten både av tilstand (2) og av tilstand (6) i D-automaten på side 41.

For å kunne nyttiggjøre oss klarsymbolet beregner vi først etterfølgermengden til hvert av metasymbolene. Hvordan dette gjøres er forklart tidligere. For hver av D-tilstandene som ikke tilfredstiller LR(0)-kravet gjør vi så følgende: Bak hvert av reduksjonselementene setter vi opp etterfølgermengden til det metasymbolet som utgjør venstresiden i elementet.

Om det finnes leseelementer i D-tilstanden så danner vi en *lese-etterfølgermengde* for denne tilstanden, ved å ta *alle grunnsymboler som forekommer som merker på utgående kanter fra denne tilstanden.* Dette er altså den mengde av grunnsymboler som vi kan lese inn på stakken, og fremdeles ha en stakk der det finnes en fortsettelse som gir en riktig setning.

Dersom vi er i en gitt D-tilstand og vi har et gitt klarsymbol, kan nå fastslå følgende:

- Dersom *lesing* er den riktige operasjonen (og vi har en riktig setning) så må klarsymbolet være i lese-etterfølger-mengden til tilstanden.
- Dersom tilstanden har et visst reduksjonselement, så kan den tilsvarende reduksjonen være det rette å gjøre bare dersom klarsymbolet ligger i etterfølgermengden til dette reduksjonselemetet (eller altså til produksjonens venstreside).

SLR-kravet

SLR-kravet er nå rett og slett at for alle tilstander (og vi behøver bare å bekymre oss over de som ikke tilfredstilte LR(0)-kravet), så er disse mengdene disjunkte. Altså, etterfølgermenden til hver av reduksjonselementene er disjunkte, og unionen av disse er igjen disjunkt med lese-etterfølgermengden.

Det lar seg da entydig avgjøre hva som er den rette aksjon å foreta seg, ved bare å se på topp-tilstanden og klar-symbolet. Dersom dette er i lese-etterfølger-mengden så må lesing være den rette operasjon, mens om det er i etterfølger-mengden til et av reduksjonselementene, så er det denne reduksjonen som er den rette. Vi kan derved igjen entydig finne høyre-syntaks-parentesene, og vi har en parseringsalgoritme. Denne algoritmen kalles SLR-algoritmen (SLR står for "Simple LR"), og grammatikker der D-automaten tilfredstiller disjunthetskravet over kalles SLR-grammatikker.

Denne algoritmen kan passelig fungere ut fra en to-dimensjonal tabell over hva som skal gjøres for hver tilstand og for hvert klar-symbol. Dette skal vi kommentere litt om et øyeblikk, og vi skal se på det i oppgaver.

SLR-algoritmen finner også feil "så fort som mulig"

Et viktig spørsmål blir nå hvordan denne SLR-algoritmen vil reagere på setninger som er gale. Om vi antar at en grammatikk tilfredstiller SLR-kravet, vil SLR-algoritmen da på en passelig måte påvise feilen i setninger som ikke ligger i grammatikkens språk?

Svaret er at det vil den, og nøkkelen ligger igjen i at vi aldri vil lese et symbol over på stakken med mindre stakken godkjennes av automaten, og derved kan være starten på en setning i språket. Alle de reduksjonene vi gjør vil stamme fra reduksjons-elementer i topptilstanden, og vil derfor (slik vi tidligere påstod, uten bevis) føre oss over i en ny lovlig stakk. En eller annen gang må det imidlertid bli snakk om å lese, og senest da vil det bli oppdaget dersom klarsymbolet er en umulig fortsettelse av det vi har lest.

I denne forbindelse kan vi registrere at vi har visse valg med hensyn til hvordan algoritmen skal reagere. Fordelen med denne valgsitasjonen i forhold til de fleste andre er imidlertid at *samme hva du gjør innenfor rimelighetens grenser, så går det bra.* Og rimelighetens grenser går nettopp ved *aldri å lese klarsymbolet over på stakken uten at det er med i lese-etterfølgermengden til topptilstanden.* For å kunne lese må vi altså forlange at det går en kant merket med klarsymbolet fra topptilstanden.

Når det gjelder reduksjoner kan vi imidlertid være mer eller mindre kritiske. I en D-tilstand som tilfredstiller LR(0)-kravet ved å ha ett reduksjons-element kan det f.eks. være naturlig å gjøre reduksjonen uten å se på klarsymbolet i det hele tatt (som i LR(0)-algoritmen). På den annen side kunne vi jo allerede her si at det må være noe galt dersom klarsymbolet ikke er med i etterfølgermengden til metasymbolet på venstresiden i reduksjons-elementet.

Generelt kan vi definere en "total etterfølgermengde" for hver D-tilstand, som unionen av dens lese-etterfølgermengde og etterfølgermengdene til hver av (venstresidene til) reduksjons-elementene i tilstanden. Om vi vil være så kritiske som mulig, kan vi signalere en feil så fort klarsymbolet ikke er med i den totale etterfølgermengden til topptilstanden.

Implementasjon av SLR-algoritmen

Avslutningen for SLR-algoritmen kan foregå på samme måten som for LR(0)-algoritmen diskutert tidligere, og vi skal holde oss til det. Alternativt kunne vi avslutte parseringen ett hakk tidligere, nemlig når vi er i en tilstand som inneholder elementet "S' → S . @", og klarsymbolet er '@'. I mange fremstillinger gjøres det på denne siste måten, og fordelen er at man da får en tilstand mindre å lagre i tabellverk etc.

Selve implementasjonen av algoritmen kan vi selvfølgelig gjøre ved et eksplisitt program, slik vi gjorde for LR(0)-eksempelet tidligere. Det er imidlertid vanligere å legge all informasjon som har med LR-parsering av en gitt grammatikk i en tabell, å så ha en fast algoritme som arbeider ut fra denne tabellen. Hvordan vi kan sette opp en slik tabell, og hvordan vi skal bruke den under parseringen, skal vi se på i forbindelse med vår eksempelgrammatikk i neste avsnitt. Grovt sett vil tabellen ha en linje for hver tilstand i D-automaten, og en kolonne for hvert mulig klarsymbol, og for en gitt topptilstand og et gitt klarsymbol vil det i den tilsvarende ruten stå hvilken aksjon som er den riktige.

Hver linje representerer en tilstand i D-automaten, og hver kolonne representerer en

Om vi har funnet at vår grammatikk tilfredstiller SLR-kravet, så er det rimelig å lage et tabellverk som er noe utvidet i forhold til det som er nødvendig for LR(0)-algoritmen. Selve D-automatene må vi lagre som før, men det er ikke lenger bare én mulig aksjon for hver tilstand. I stedet er aksjonen avhengig av klarsymbolet, så vi må for hver tilstand ha en tabell over hva vi skal gjøre for hvert mulig klarsymbol. For de klarsymboler som ikke er med i tilstandens totale etterfølgermengde kan vi umiddelbart rapportere en feil. Ellers skal vi rapportere at setningen er OK i avslutningssituasjonen angitt over, og ellers skal vi enten angi en leseoperasjon, eller en bestemt reduksjon.

Vi kan også lage en tabell som frigjør oss mer fra den tegnede D-automaten, ved også å angi de nye tilstander i tabellen. Hvordan dette kan gjøres skal vi se på i oppgaver.

Vår eksempelgrammatikk tilfredsstiller SLR-kravet

Om vi vil undersøke om vår eksempel-grammatikk tilfredstiller SLR-kravet, må vi først se på etterfølgermengdene for hver av metasymbolene. Den beregnet vi i kapittel 3, og fant der følgende:

For U(TTRYKK): +, @
For T(ERM) : +, ∗, @

Om vi ut fra dette studerer tilstandene (2) og (6) i automaten på side 41, ser vi at disjunkthetskravet er oppfylt. Vi vet dermed alltid hvilken aksjon som er den rette: klarsymbolet:

Tilstand (2):

U → T .	Denne reduksjonen velges dersom klarsymbolet er: +, @
U → T . ∗ **navn**	Lesing velges dersom klarsymbolet er: ∗

Tilstand (6):

U → U + T .	Denne reduksjonen velges dersom klarsymbolet er: +, @
U → T . ∗ **navn**	Lesing velges dersom klarsymbolet er: ∗

Vi lar det være en øvelse å sette opp en passelig tabell for dette.

4.3.6 Litt om andre typer LR-grammatikker

.... Ta inn tabell fra "oppgave 0" i oppgavekompendium her .....

KOPIERT FRA OPPGAVEKOMPENDIUM:

Tilstand	Klarsymbol				Ny tilst. ved red.	
	navn	+	∗	@	U	T
0	les til 3	feil	feil	feil	1	2
1	feil	les til 4	feil	les til 8	–	–
2	feil	red. U → T	les til 5	red. U → T	–	–
3	red. T → **navn**	red. T → **navn**	red. T → **navn**	red. T → **navn**	–	–
4	les til 3	feil	feil	feil	–	6
5	les til 7	feil	feil	feil	–	–
6	feil	red. U → U+T	les til 5	red. U → U+T	–	–
7	red. T → T∗**navn**	red. T → T∗**navn**	red. T → T∗**navn**	red. T → T∗**navn**	–	–
8	ferdig	ferdig	ferdig	ferdig	–	–

Vi kunne her alternativt sløyfe tilstand 8, og i stedet sette “ferdig” i tilstand 1, kolonne ‘@’.

Som en trening på bruk av denne, kan man kjøre LR-analyse av eksempelet umiddelbart før kapittel 4.3.1 og sjekke at man får de riktige aksjoner.

@@■

Mange grammatikker som oppstår i forbindelse med praktiske programmeringsspråk er SLR-grammatikker. Vi skal imidlertid legge merke til at det for et gitt språk kan formuleres mange grammatikker, og det er ofte slik at noen av disse tilfredstiller SLR-kravet, mens andre ikke gjør det. Det kan derfor kreves litt fomling før man finner fram til en formulering som tilfredstiller kravene.

Det finnes imidlertid mange språk som ikke kan gis *noen* grammatikk som er SLR, og også for vanlige programmeringsspråk finnes visse konstruksjoner som gjør det vanskelig eller umulig å formulere SLR-grammatikker.

LALR-grammatikker

En forholdsvis grei måte å gå videre på er å forsøke å gjøre en mer nøyaktig analyse av hva som faktisk kan følge etter de forskjellige elementer i de forskjellige tilstander av D-automaten. Det vi gjorde over, dvs. å ta etterfølgermengden til venstresiden, viser seg å være en ganske grov analyse. Ved å studere D-automaten nøye kan man ofte få redusert etterfølgermengdene for reduksjonselemenene i noen av D-tilstandene.

Dette gir da nytt håp om at de forskjellige etterfølger-mengdene innen hver tilstand kan være disjunkte, og grammatikker der vi kan oppnå disjunkthet på denne måten sies å være

LALR. Det er faktisk slik at de aller fleste konstruksjoner i programmeringsspråk greit kan beskrives med grammatikker som tilfredstiller LALR-kravet. Dog må det ofte noe omskriving til før man kommer i mål.

De fleste systemer som automatisk genererer analysatorer (parsere) ut fra gitte grammatikker er laget etter dette prinsippet. Et kjent slik system er YACC (Yet Another Compiler-Compiler), som finnes på de fleste UNIX-systemer.

LR(1)- og LR(k)-grammatikker

Heller ikke ved LALR-teknikk klarer man imidlertid å lage analysatorer som virker for alle grammatikker som ut fra teoretiske betrakninger er LR(1) (nemlig de der man ved å titte på klarsymbolet løpende har *informasjon nok* til å gjøre LR-parsering). For å lage en generell LR(1)-analysator må man ta utgangspunkt i det faktum at de mulige stakker *utvidet med de mulige klarsymboler* også danner et regulært språk, og gjøre alt om igjen ut fra dette.

Det er ikke noe problem å gjøre dette, og om man *gjør* det har man et apparat som ville ta de aller fleste grammatikker uten omskrivninger. Problemet er imidlertid at dette fører til mange flere tilstander, og da blir straks tabellverkene svært store og uhåndterlige. For et vanlig programmeringsspråk kan det lett føre til at man får mange tusen tilstander, mens man ved SLR- eller LALR-betrakninger gjerne får noen få hundre tilstander, og dette gjør all verdens forskjell for en effektiv implementasjon.

Man kan også utvide disse betrakningene videre ved å si at vi ikke bare ser på klarsymbolet, men på de k neste symbolene. Teorien blir da ikke særlig mye verre, men antallet tilstander vil derimot øke dramatisk. Derfor er dette lite brukt i praksis.

4.3.7 Presedens og flertydige grammatikker

La oss tenke oss at vi har fått i oppdrag å regne ut "$5 + 4 * 6$". De fleste vil få svaret 29 og ikke 54. Vi tolker altså uttrykket som "$5 + (4 * 6)$". Dette skyldes neppe at vi har den entydige grammatikken fra kapittel 2 (side 8) i hode og derfor setter inn syntaksparentesene i henhold til denne. Årsaken er heller at vi på barneskolen lærte at multiplikasjon "binder sterkere" enn addisjon. Med en teknisk term sier vi at operatoren '$*$' *har høyere presedens* enn operatoren '$+$'. Det trenger imidlertid ikke å være to *forskjellige* operatorer involvert for at slike problemer skal oppstå. I uttrykket "$4 - 3 - 5$" er det f.eks. underforstått at det venstre minustegnet binder sterkere enn det høyre. Vi tolker det som "$(4 - 3) - 5$" og ikke som "$4 - (3 - 5)$". I dette tilfellet sier vi at subtraksjonsoperatoren er *venstreassosiativ*.

Matematisk notasjon er full av slike bindingsregler. I det følgende skal vi se litt på hvordan en flertydig grammatikk G kan suppleres med regler for presedens og assosiativitet slik at hver setning får en entydig tolkning, og slik at vi forholdsvis greit kan lage en syntaksanalysator av LR-typen for G (utvidet med disse reglene).

Vi har i det foregående konsentrert oss om entydige grammatikker, og en grunn til dette er at flertydige grammatikker aldri kan tilfredstille noe LL-krav eller LR-krav (rett og slett fordi det ikke er bestemt hvor syntaksparentesene skal stå, selv om vi kan se hele resten av setningen). Det er imidlertid slik at en flertydig grammatikk ofte har færre produksjoner og færre metasymboler enn en entydig grammatikk for det samme språket, og den kan derfor være mer oversiktlig enn en entydig grammatikk. Et enkelt eksempel på dette er følgende flertydige og entydige grammatikk for aritmetriske uttrykk med addisjon og multiplikasjon:

Flertydig grammatikk:

$$
\begin{array}{lcl}
U & \rightarrow & U + U \\
U & \rightarrow & U * U \\
U & \rightarrow & \textbf{navn}
\end{array}
$$

Entydig grammatikk:

$$
\begin{array}{lcl}
U & \rightarrow & U + T \\
U & \rightarrow & T \\
T & \rightarrow & T * \textbf{navn} \\
T & \rightarrow & \textbf{navn}
\end{array}
$$

I eksemplet over er begge grammatikkene rimelig håndterlige, men for mer kompliserte språk kan gevinsten ved å bruke flertydige grammatikker være betydelig. For å kunne hanskes med en flertydig grammatikk trenger man tilleggsregler for å løse tolkningskonflikter. En type slike tilleggsregler er de tidligere nevnte presedensreglene.

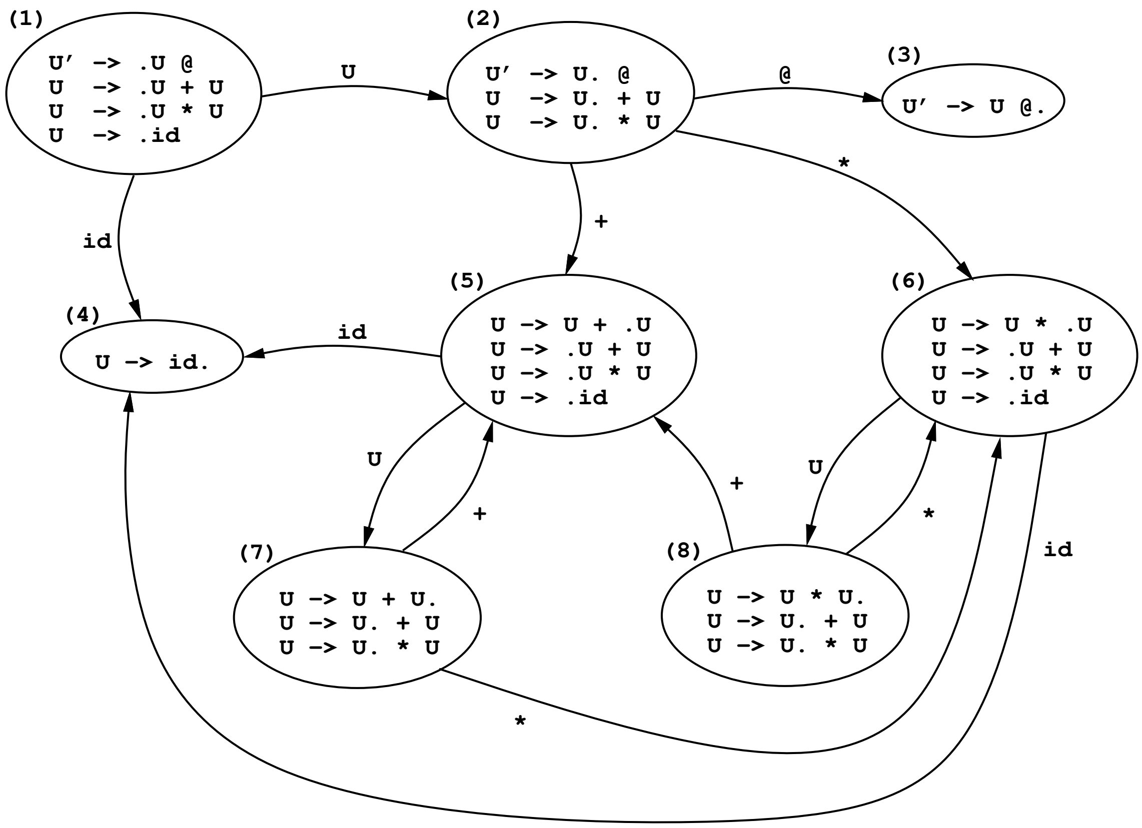
I den entydige grammatikken over er reglene for presedens innbakt i produksjonene, mens det i den flertydige grammatikken ikke finnes regler for presedens. En mulig fremgangsmåte for å hanskes med tolkningskonfliktene i den flertydige grammatikken er å konstruere den deterministiske LR(0)-automaten som før, og deretter løse de konflikter som må oppstå ved hjelp av de vanlige reglene for presedens og assosiativitet.

Vi skal nå skissere hvordan dette gjøres for grammatikken angitt over, men man skal være klar over at det teoretiske fundament for dette, samt spørsmålet om når dette vil fungere greit, er nokså komplisert. Fremstillingen under får derfor et litt ad.hoc.-preg, men tilsvarende teknikk kan i praksis brukes for å løse mange vanlig forkommende syntaksproblemer. Dette gjelder spesielt i syntaksen for forskjellige typer uttrykk, men teknikken kan f.eks. også brukes til å løse det såkalte "dangling else" problemet.

Vi utvider først den flertydige grammatikken på standard måte med et nytt startsymbol og får grammatikken

$$
\begin{array}{lcl}
U' & \rightarrow & U\ @ \\
U & \rightarrow & U + U \\
U & \rightarrow & U * U \\
U & \rightarrow & \textbf{navn}
\end{array}
$$

Selv om dette ikke er en entydig grammatikk vil innholdet av stakken fremdeles kunne beskrives av en endelig automat. Den deterministiske automaten for grammatikken over er vist under.



Vi ser at i automaten materialiserer flertydigheten seg som les/reduser-konflikter i tilstandene 7 og 8. I begge er det ett reduksjonselement og to leselementer. Disse konfliktene kan ikke løses på SLR-manér siden både '∗' og '+' er i etterfølgermengden til `U`. Faktisk vil enhver LR(k)-analysator for grammatikken ha disse konfliktene.

La oss først se på de mulige konfliktsituasjonene for tilstand 7. Elementet "U → U + U." forteller oss at vi med denne topptilstanden må ha "`U + U`" på toppen av stakken. Ta derfor som eksempel setningen "**navn** + **navn** ∗ **navn**". Etter at "**navn** + **navn**" er lest (og redusert) vil vi ha situasjonen:

$$\underbrace{1\ \mathrm{U}\ 2\ +\ 5\ \mathrm{U}\ 7}_{\text{Stakk}} \qquad\qquad \underbrace{*\ \textbf{navn}\ @}_{\text{Ulest}}$$

Siden '∗' binder sterkere enn '+' skal vi nå ikke redusere, men flytte '∗' over på stakken. Også setningen "**navn** + **navn** + **navn**" vil føre til konflikt i tilstand 7. Etter at "**navn** + **navn**" er lest vil vi ha situasjonen:

$$\underbrace{1\ \mathrm{U}\ 2\ +\ 5\ \mathrm{U}\ 7}_{\text{Stakk}} \qquad\qquad \underbrace{+\ \textbf{navn}\ @}_{\text{Ulest}}$$

Hvis vi velger å redusere i dette tilfellet, vil setningen tolkes som "(**navn** + **navn**) + **navn**". Velger vi derimot å flytte '+' over på stakken vil setningen bli tolket som "**navn** + (**navn** + **navn**)".

I det første tilfellet tolkes '+' som en *venstreassosiativ* operator, mens i det andre tilfellet tolkes den som en *høyreassosiativ* operator. For addisjon av enkle variabler spiller dette ingen rolle, men denne distinksjonen blir essensiell når vi vil ha med subtraksjon (som syntaktisk rimeligvis bør behandles nøyaktig som '+').

Konfliktene i forbindelse med tilstand 8 løses på samme vis. For at en SLR-analysator skal kunne fungere for grammatikken over må det altså opplyses om presedensreglene, dvs. om '$*$' tar presedens over '+' og om '$*$' og '+' er høyre- eller venstre-assosiative. Dersom vi, som i matematikken, lar '$*$' ha presedens over '+' og om begge operatoren er venstreassosiative, får vi følgende entydige regler for tilstandene 7 og 8:

Tilstand 7: Lesing om klarsymbolet er '$*$', ellers reduksjon i henhold til "U $\rightarrow$ U + U".

Tilstand 8: Alltid reduksjon i henhold til "U $\rightarrow$ U $*$ U".

Som nevnt tidligere finnes det systemer som genererer analysatorer ut fra gitte grammatikker. I forbindelse med slike systemer er det som oftest mulig å bruke flertydige grammatikker supplert med regler for presedens og assosiativitet. Dette er f.eks. tilfellet for systemet YACC.

.... Ta inn tabell fra "oppgave 1" i oppgavekompendium her .....

KOPIERT FRA OPPGAVEKOMPENDIUM:

Siden multiplikasjon har høyere presedens enn addisjon skal det i tilstand 7, når klarsymbolet er '$*$', leses og ikke reduseres. Ellers skal det reduseres så tidlig som mulig, fordi '+' og '$*$' er venstreassosiative.

Analysetabell med utgangspunkt i automaten i kapittel 4.3.7:

Tilstand	Klarsymbol				Ny tilst. ved red.
	id	+	$*$	@	U
1	les, til 4	feil	feil	feil	2
2	feil	les, til 5	les, til 6	les, til 3	–
3	ferdig	ferdig	ferdig	ferdig	–
4	$*$	red. U $\rightarrow$ **id**	red. U $\rightarrow$ **id**	red. U $\rightarrow$ **id**	–
5	les, til 4	feil	feil	feil	7
6	les, til 4	feil	feil	feil	8
7	$*$	red. U $\rightarrow$ U+U	les, til 6	red. U $\rightarrow$ U+U	–
8	$*$	red. U $\rightarrow$ U$*$U	red. U $\rightarrow$ U$*$U	red. U $\rightarrow$ U$*$U	–

'$*$' betyr at reduksjon kan gjøres, men det er ikke nødvendig fordi det vil bli feil senere.

Man kunne også skrevet "ferdig" i tilstand 2, kolonne '@', og kuttet ut hele 'tilstand 3'.

@@@

Øvelse 1

Lag en analysetabell (parsing table) for grammatikken over, dvs angi hva som skal gjøres for hver topptilstand og hvert klarsymbol. Forsøk også å få inn i tabellen hva den nye topptilstanden skal være. Anta at '$*$' har høyere presedens enn '$+$' og at begge operatorene er venstreassosiative.

Øvelse 2

Sett opp D-automaten og gi et sett med presedensregler som løser konfliktene for grammatikken:

$$
\begin{array}{lcl}
\mathrm{U} & \rightarrow & \mathrm{U} + \mathrm{U} \\
\mathrm{U} & \rightarrow & \mathrm{U} * \mathrm{U} \\
\mathrm{U} & \rightarrow & (\ \mathrm{U}\ ) \\
\mathrm{U} & \rightarrow & \textbf{navn}
\end{array}
$$

Lag så en analysetabell for grammatikken i henhold til D-automaten og de vanlige presedens- og assosiativitetsregler. $\blacksquare$