```
begin { Udtryk }
   SimpeltUdtryk;
   if brik.briktype in [operator] then
      begin
         NæsteBrik(brik);
         Udtryk(brik)
      end;
end; { Udtryk }

begin { hovedprogram }
   NæsteTegn;
   NæsteBrik(brik) ;
   Udtryk(brik)
end. { hovedprogram }
```

Eksempel 2.11 EBNF-grammatikken for non-terminalen *udtryk* kan også skrives:

```
udtryk         = [udtryk  operator] simpelt-udtryk.
simpelt-udtryk = osv.
```

Produktionsreglen for *udtryk* er **venstre-rekursiv**, dvs. *udtryk* optræder helt til *venstre* i højresiden i produktionsreglen for *udtryk*.

Vi vil igen gerne kontrollere, om en given følge af data overholder EBNF-grammatikken ovenfor. Den eneste ændring, der er nødvendig i programmet i eksempel 2.10, er at udskifte underprogrammet *Udtryk* med følgende underprogram:

```
procedure Udtryk(brik: brikpost);
begin { Udtryk }
   if brik.briktype in [positivtheltal, højreparentes] then
      begin
         Udtryk(brik);
         OperatorProcedure
      end;
   SimpeltUdtryk
end; { Udtryk }
```

Når syntakskontrollen sættes igang vil den første brik af en udtryk blive læst ind i variablen *brik*. Hvis *brik.briktype* indeholder et first-symbol for *udtryk*, vil underprogrammet *Udtryk* kalde sig selv. Underprogrammet *Udtryk* vil endnu engang undersøge variablen *brik.briktype* og kalde sig selv osv. Vi har altså en uendelig løkke. Hvis vi havde undersøgt om krav 2 fra afsnit 3.2 var opfyldt, havde vi fundet ud af, at dette ikke var tilfældet:

first([udtryk operator]) ∩ follow([udtryk operator]) =
first(udtryk) ∩ first(simpelt-udtryk) =
first(simpelt-udtryk) ∩ first(simpelt-udtryk) =
{positivt-heltal, venstreparentes} ∩ {positivt-heltal, venstre-parentes} <>
∅.

Der gælder generelt, at regelstyret indlæsning *ikke* kan anvendes i forbindelse med *venstre-rekursive* produktionsregler i EBNF-grammatikker. Til gengæld findes der algoritmer til at omskrive venstrerekursive EBNF-grammatikker til ækvivalente uden venstrerekursion. For en nærmere uddybning af dette og beslægtede emner se [8].

2.4 Tabelstyret indlæsning

I dette afsnit introduceres en anden metode, der kan kontrollere, om de indlæste brikker stemmer overens med en given EBNF–grammatik, samt uddrage betydningen af brikkerne sat i relation til hinanden. Metoden hedder tabelstyret indlæsning. Afsnittet bruger de procedure- og variabelnavne, der blev defineret i afsnit 2.

Tabelstyret indlæsning benytter sig, som det fremgår af navnet, af tabeller. Disse tabeller benævnes *tilstandstabel* og *aktionstabel*. Tabelstyret indlæsning bruger som regelstyret indlæsning *single-symbol-lookahead* strategien. Vi vil i de følgende afsnit beskrive, hvad en *tilstand* og en *aktion* er, samt beskrive hvordan en *tilstandstabel* og en *aktionstabel* opbygges. Dette gøres ud fra et sammenhængende eksempel. Herefter findes et eksempel på, hvordan tabelstyret indlæsning programmeres. Tilsidst vil vi beskrive, hvilke krav EBNF–grammatikken skal opfylde, for at man kan benytte tabelstyret indlæsning.

2.4.1 Tilstande og aktioner

For underprogrammer, der foretager indlæsning, gælder at de data, der indlæses, indvirker på, hvilke programlinier, der udføres. gælder, at de programlinier, der er udført, afhænger af, hvordan inddata ser ud. Man kan således sige, at et program er i en bestemt tilstand alt efter, hvilke programlinier, der hidtil har været udført, samt efter hvilke inddata, der indlæses. Programmets tilstand er således en funktion – vi kalder den her F – af de allerede udførte programlinier og de allerede indlæste inddata. Eller udtrykt på en anden måde:

```
aktuelle tilstand = F(allerede udførte linier, inddata)
```

En *ny* tilstand kan nu findes som en funktion, – vi kalder den her F1 – af den aktuelle tilstand og de netop indlæste inddata. Dette vil vi kalde *tilstandsovergang* og udtrykke på følgende måde:

```
nytilstand = F1(aktuelle tilstand, inddata)
```

En aktion er en operation, der skal udføres afhængig af inddata og den aktuelle tilstand. Eller udtrykt som en funktion, F2, af den aktuelle tilstand og de netop indlæste inddata:

```
aktion = F2(aktuelle tilstand, inddata)
```

Som det ses, kan vi – afhængig af den aktuelle tilstand og inddata – føres over i en ny tilstand og altså videre i programmet. På samme måde bestemmer en tilstand sammen med inddata, hvilken aktion programmet skal udføre. Man kan altså **befinde** sig i en tilstand, mens man kan **udføre** en aktion. Det at udføre aktionen har ikke indflydelse på tilstanden, hvorimod tilstanden sammen med inddata er bestemmende for aktionen.

I tabelstyret indlæsning opretter man, som nævnt, en *tilstandstabel* og en *aktionstabel*. *Tilstandstabellen* indeholder alle relevante tilstandsovergange ved forskellige typer af inddata, mens *aktionstabellen* indeholder alle relevante aktioner. Aktionerne kan udføres i forskellige tilstande afhængig af nyindkomne inddata. Man kan nu i programmet bruge disse tabeller til:

1. at finde den nye tilstand, programmet skal fortsætte i.
2. at udføre den korrekte aktion.

Vi vil nu i det følgende forklare, hvordan disse to tabeller konstrueres.

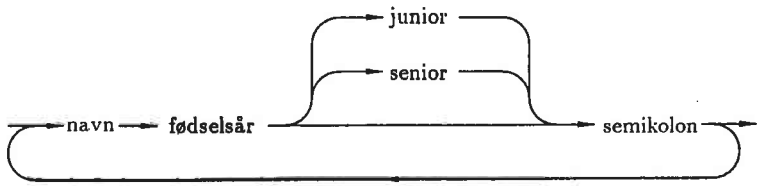
2.4.2 Tilstandstabel

Vi betragter igen den konkretiserede EBNF–grammatik fra afsnit 2.1:

Eksempel 2.12

```
medlemsliste = medlem {medlem}.
medlem       = navn fødselsår [kategori] semikolon.
kategori     = junior | senior.
```

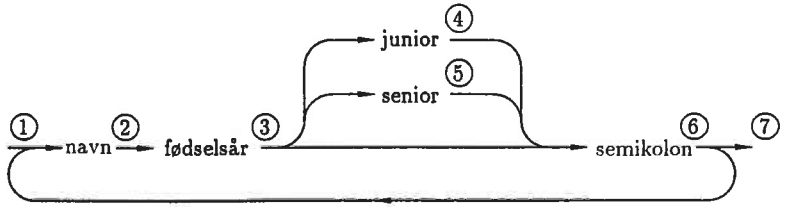
Som det ses, kan fx *navn* optræde i starten men ikke efter kategori. Samme brik skal således behandles forskelligt, alt efter hvor i indlæsningen vi befinder os. Det er derfor vigtigt at vide, hvilken tilstand indlæsningen befinder sig i. Vi omskriver vores EBNF–grammatik til en syntaksgraf (jvf. kap. 1.5). Idet vi kun opskriver brikkerne fås følgende syntaksgraf:



Vi kan nu definere en tilstand:

Definition 2.4 En **tilstand** svarer til udgangspunktet for en pil i syntaksgrafen.

Hvis vi ud fra vores definition indsætter tilstandene får vi:



Vi har her tilføjet en sluttilstand (tilstand nr 7). Indlæsningen er afsluttet, når man kommer i denne tilstand.

Det ses fx, at hvis vi befinder os i tilstand (1) er *navn* tilladt, mens vi i tilstand (2) ikke tillader *navn*. Hvis vi i tilstand (2) møder *navn*, skal vi fortsætte i en "fejltilstand", hvor vi fx kan definere, at der skal udskrives en fejlmeddelelse og derefter fortsættes i en slut tilstand. Vi kan tildele hver af tilstandene et sigende navn og har nu følgende 8 tilstande:

1. starttilstand
2. efter-navn
3. efter-fødselsår
4. efter-senior
5. efter-junior
6. efter-semikolon
7. sluttilstand
8. fejltilstand

Meningen med den tabel, vi nu vil opbygge, er, at vi i den aktuelle tilstand ud fra brikkerne skal afgøre, hvilken ny tilstand vi skal fortsætte i. Hvis vi fx befinder os i tilstanden efter-fødselsår, så tillader vi:

- *junior*, der bringer os til efter-junior
- *senior*, der bringer os til efter-senior
- *semikolon*, der bringer os til efter-semikolon

Ved alle andre brikker skal vi fortsætte i fejltilstanden. På syntaksgrafen ses, at vi både i tilstanden efter-senior og efter-junior kun tillader *semikolon*. Det vil derfor være hensigtsmæssigt at lade indlæsningen reagere ens ved både *junior* og *senior* og slå de to tilstande sammen til en fx *efter-kategori*. *Junior* og *senior* angiver nu en klasse, der kan forstås som en samling data, overfor hvilke programmet skal reagere ens. Vi kan kalde klassen for *kategori*. Vi har nu følgende forskellige muligheder for inddata-typer, klasser, (jvf afsnit 2):

- navn
- fødselsår
- kategori
- semikolon
- slut
- andet

Som grundlag for den endelige tilstandstabel opskriver vi en oversigt over hhv. tilstande, de tilladte klasser/ikke–tilladte klasser samt de nye tilstande, som disse klasser fører over i. Vi begynder i starttilstanden og definerer, hvilke klasser vi tillader og i hvilken tilstand, vi skal fortsætte, samt definerer, hvilke klasser vi ikke tillader og i hvilken tilstand vi her skal fortsætte. Fx tillader vi *navn* i starttilstanden, hvilket bringer os videre i tilstand efter-navn, mens alle andre klasser er ulovlige og bringer os i fejltilstanden. Herefter fortsætter vi opbygningen af tabellen med i tilstanden efter-navn at definere, hvad der hhv. tillades, og ikke tillades og i hvilken tilstand, der fortsættes osv. Resultatet ses i figur 2.1 side 65.

Som det ses, forekommer der et spørgsmålstegn ved sluttilstand og fejltilstand. Man må i de enkelte programmer tage stilling til, hvad der skal ske.

I vores eksempel definerer vi, at det i sluttilstand er underordnet, hvilken tilstand vi fortsætter i. Til at angive dette har vi valgt '?'.
I efter-semikolon har vi to muligheder:

1. alt undtagen klassen navn skal være tilladt for at komme videre i sluttilstand eller
2. kun sluttypen skal være tilladt for at komme videre i sluttilstand

gl. tilstand	klasse	nye tilstand
start-tilstand	navn tilladt alt andet ikke tilladt	efter-navn fejltilstand
efter-navn	fødselsår tilladt alt andet ikke tilladt	efter-fødsel fejltilstand
efter-fødselsår	kategori tilladt semikolon tilladt alt andet ikke tilladt	efter-kategori eft-semikolon fejltilstand
efter-semikolon	navn tilladt alt andet tilladt	efter-navn sluttilstand
sluttilstand	alt tilladt	?
fejltilstand	afhænger af definition	?

Figur 2.1: Oversigt over tilstandsovergange

Vi har valgt mulighed (1). Endvidere har vi defineret, at vi i fejltilstand altid fortsætter i sluttilstand.

Vi vil nu opskrive en tilstandstabel ud fra tilstandsovergangene. De vandrette indgange er tilstandene, mens de lodrette indgange er klasserne. I felterne angives, hvilken ny tilstand man skal fortsætte i, når man i en given tilstand møder den pågældende klasse af inddata. Tilstandstabellen findes i fig. 2.2 side 66 og følgende forkortelse er brugt:

```
start    = starttilstand
eft-navn = efter-navn
eft-f-år = efter-fødselsår
eft-kat  = efter-kategori
eft-semi = efter-semikolon
slut     = sluttilstand
fejl     = fejltilstand
```

Tabellen kan i Pascal defineres som:

```
type
   brikker   = (navn, fødselsår, kategori, semikolon, slut, andet);
   tilstande = (StartTilstand, EfterNavn, EfterFødselsår,
                EfterKategori, EfterSemikolon, SlutTilstand,
                FejlTilstand);
var
   tilstandstabel:array[Starttilstand..EfterSemikolon,brikker] of tilstande;
   klasse:brikker;
```

Blanke felter angiver fejltilstand.

	navn	fødsels-år	kategori	semi-kolon	slut	andet
starttilstand	eft-navn					
efter-navn		eft-f-år				
efter-fødselsår			eft-kat	eft-semi		
efter-kategori				eft-semi		
efter-semikolon	eft-navn	slut	slut	slut	slut	slut

Figur 2.2: Tilstandstabel

Sluttilstand og fejltilstand er ikke taget med i tabellen, da der i disse tilstande ikke skal foretages tilstandsskift afhængigt af inddata.

2.4.3 Aktion

Vi har i dette afsnit indtil videre udelukkende koncentreret os om at kontrollere, at det, der indlæses, er i overensstemmelse med en given EBNF–grammatik. Men de fleste programmer skal også udføre nogle aktioner løbende med syntakskontrollen. I vores eksempel kunne man fx forestille sig, at inddata skulle indlæses i en post, der skulle indgå i en database. Vi kan derfor vælge, at vi i tilfælde af fejl ønsker at udskrive en fejlmeddelse. Endvidere kan man forestille sig, at vi efter indlæsningen af en kategori ønsker at kontrollere, om det semantiske krav, vi stillede i kapitel 1 afsnit 3, bliver overholdt. Dvs, at *junior* skal være født mellem '70 og '88, mens *senior* skal være født mellem ex. '00 og '69. Desuden kan nogle aktioner være at gemme værdierne af *navn*, *fødselsår* og *kategori* i nogle variable efterhånden, som de indlæses, og endelig kan vi have en slutaktion. Vi kan opskrive følgende aktioner:

- gem-navn
- gem-fødselsår
- gem-kategori
- check-kategori
- fejlmeddelelse
- slut

Ud fra ovenstående kan vi opstille en skitse over aktionernes indhold:

```
check-kategori :  if kategori = junior then ...
gem-navn       :  navne = brik.lexem
gem-fødselsår  :  fødselsår = brik.lexem
gem-kategori   :  kategori = brik.lexem
```



```
check-kategori :  if kategori = junior then ...
fejlmeddelse   :  Fejl(...)
slut           :  slut(...)
```

Valget af aktion skal ske ud fra klassen. Men som nævnt i afsnit 2.4.2, forekommer det, at samme klasse skal behandles forskelligt forskellige steder i indlæsningen. Så vi har brug for at vide, hvilken tilstand indlæsningen befinder sig i. Fx skal vi i starttilstand gemme *navn* i variablen *navne*, mens vi i tilstanden efter-kategori skal udskrive en fejlmeddelse, hvis næste klasse er *navn*. Vi kan nu, ligesom vi gjorde det ved tilstandstabellen, systematisk gå de enkelte tilstande igennem og opskrive, hvilke aktioner der skal udføres, og hvilke klasser der er lovlige. Opbygningen er den samme som opbygningen af tabellen i fig. 2.1. Vi starter igen i tilstanden starttilstand og kommer frem til figur 2.3.

gl. tilstand	klasse	aktion
start-tilstand	navn tilladt alt andet ikke tilladt	navneaktion fejlaktion
efter-navn	fødselsår tilladt alt andet ikke tilladt	fødselsåraktion fejlaktion
efter-fødselsår	kategori tilladt semikolon tilladt alt andet ikke tilladt	kategoriaktion semikolonaktion fejlaktion
efter-semikolon	navn tilladt alt andet tilladt	navneaktion fejlaktion
sluttilstand	alt tilladt	slutaktion

Figur 2.3: Oversigt over aktioner

Aktionstabellen kan opskrives på grundlag af fig. 2.3. De vandrette og lodrette indgange er igen hhv. tilstandene og klasserne, mens felterne angiver, hvilken aktion der skal udføres, når man i den givne tilstand møder den pågældende klasse af inddata. Aktionstabellen findes i fig. 2.4, idet følgende forkortelser bruges:

```
navne-akt = navneaktion
f-år-akt  = fødselsåraktion
kat-akt   = kategoriaktion
semi-akt  = semikolonaktion
slut-akt  = slutaktion
```

Blanke felter angiver "fejlaktion".

	navn	fødsels-år	kategori	semi-kolon	slut	andet
starttilstand	navne-akt					
efter-navn		f-år-akt				
efter-fødselsår			kat-akt	semi-akt		
efter-semikolon	navne-akt					

Figur 2.4: Aktionstabel

Sluttilstand og fejltilstand er ikke taget med i tabellen, da der i disse tilstande ikke udføres nogen aktion afhængig af ny-indlæst inddata.

Dette kan i Pascal defineres som:

```
type
    tilstande= (StartTilstand, EfterNavn, Efterfødselsår,
                EfterKategori, EfterSemikolon, SlutTilstand, Fejltilstand);
    brikker  = (navn, fødselsår, kategori, semikolon, slut, andet);
    aktioner = (NavneAktion, FødselsårAktion, KategoriAktion,
                SemikolonAktion, SlutAktion, FejlAktion);
var
    aktionstabel:array[StartTilstand..EfterSemikolon,brikker] of aktioner;
    klasse:brikker;
```

2.4.4 Brug af tilstands- og aktionstabellen

Efter nu at have defineret vores tilstandstabel og vores aktionstabel, kan vi nu gå igang med at udforme et program, der skal:

- foretage syntakskontrol
- udføre nogle forskellige aktioner

Hovedprogrammet er bygget op omkring en *repeat .. until*-løkke, samt en *case*-sætning på følgende måde:


```
...
NæsteTegn;
tilstand := StartTilstand;
repeat
   NæsteBrik(brik);
   case aktionstabel[tilstand,brik.briktype] of
      <aktion 1> : ...;
      <aktion 2> : ...;
      ...
      <aktion n> : ...;
   end;
   tilstand := tilstandstabel[tilstand,brik.briktype];
until tilstand = SlutTilstand;
```

Eksempel 2.13 Vi vil skitsere et program. Ved de enkelte aktioner udføres nogle procedurekald, som vi ikke vil beskrive nærmere. Tabellerne vil vi heller ikke initialisere fuldstændig, men blot nævne, hvordan det kan gøres.

```
Program Eksempel42;
 :
type
     brikker   = (navn, fødselsår, kategori, semikolon, slut, andet);
     tilstande = (StartTilstand, EfterNavn, EfterFødselsår,
                  EfterKategori, EfterSemikolon, SlutTilstand,
                  FejlTilstand);
     aktioner  = (NavneAktion, FødselsårAktion, KategoriAktion,
                  SemikolonAktion, SlutAktion, FejlAktion, IngenAktion);
     :
var
     tilstandstabel : array[StartTilstand..EfterSemikolon,brikker]
                                                          of tilstande;
     aktionstabel   : array[StartTilstand..EfterSemikolon,brikker]
                                                          of aktioner;
     tilstand       : tilstande;
     klasse         : brikker;
     :

     procedure Initialiser;
     begin {Initialiser}
        for tilstand := StartTilstand to EfterSemikolon do
          for klasse := navn to andet do
             begin
                tilstandstabel[tilstand,klasse] := Fejltilstand;
                aktionstabel[tilstand,klasse] := FejlAktion;
```

```
            end;
         tilstandstabel[StartTilstand,navn] := EfterNavn;
         :
         tilstandstabel[EfterSemikolon,andet] := SlutTilstand;
         aktionstabel[StartTilstand,navn] := NavneAktion;
         :
         aktionstabel[EfterSemikolon,andet] := SlutAktion
      end; {Initialiser}

      {Flere procedure- og funktionserklæringer}

begin {Hovedprogram}
   :
   NæsteTegn;
   tilstand := StartTilstand;
   repeat

      NæsteBrik(brik);
      case aktionstabel[tilstand,brik.briktype] of
         NavneAktion       : NavneProcedure(...);
         FødselsårAktion   : FødselsårProcedure(...);
         KategoriAktion    : KategoriProcedure(...);
         SemikolonAktion   : SemikolonProcedure(...);
         SlutAktion        : SlutProcedure(...);
         FejlAktion        : FejlProcedure(...);
         end;

      tilstand := tilstandstabel[tilstand,brik.briktype]
   until tilstand = SlutTilstand;
   :
end. {Hovedprogram}
```

2.4.5 Krav til EBNF–grammatikken

Kravene til EBNF–grammatikken er de samme som ved regelstyret indlæsning i afsnit 3.2. Dog kan den her nævnte metode til tabelstyret indlæsning ikke anvendes til rekursive EBNF–grammatikker. Står man over for en rekursiv EBNF–grammatik, er man nødt til at udvide metoden. Bl.a. har man brug for en *stak*. Vi vil ikke komme nærmere ind på dette i disse noter, men interesserede kan læse videre i [8]

2.5 Appendiks. Vejledende løsninger til checkopgaverne

Afsnit 2

Afsnit 2.2 Vi skal konkretisere den givne EBNF-grammatik. Som nævnt i afsnit 2.1 gøres dette ved først at fjerne alle produktionsregler, hvor en brik optræder på venstresiden. Vi fjerner altså produktionsreglerne for navn og bogstav:

```
sportsklub = medlem {medlem}.
medlem     = navn kategori ";".
kategori   = "junior" | "senior".
```

Dernæst skal alle brikker i EBNF-grammatikken, der er terminaler i den gamle EBNF-grammatik, erstattes med en briktype. Vi erstatter ";" med *semikolon*, "junior" med *junior* og "senior" med *senior*:

```
sportsklub = medlem {medlem}.
medlem     = navn kategori semikolon.
kategori   = junior | senior.
```

Afsnit 3

Afsnit 3.1

1. (a) Vi starter med at finde first-symbolerne bottom-up:

   first(navn) = {navn}
   first(junior) = {junior}
   first(senior) = {senior}
   first(semikolon) = {semikolon}
   first(medlem) = {navn}
   first(sportsklub) = {navn}

   Derefter finder vi follow-symbolerne top-down:

   follow(sportsklub) = ∅
   follow(medlem) = {navn}
   follow(navn) = {junior, senior}
   follow(junior) = {semikolon}
   follow(senior) = {semikolon}
   follow(semikolon) = {navn}