BOBS-SYSTEM

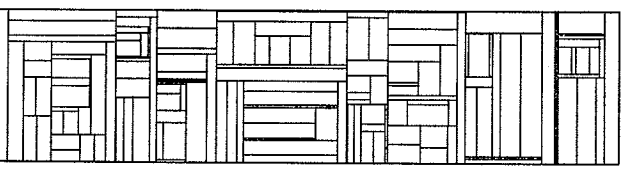
Brugervejledning

by

Bent Bruun Kristensen
Ole Lehrmann Madsen
Bent Bæk Jensen
Søren Henrik Eriksen

DAIMI PB-10
Marts 1973

Matematisk Institut Aarhus Universitet
DATALOGISK AFDELING
Ny Munkegade - 8000 Aarhus C - Danmark
Tlf. 06-12 83 55



BOBS-SYSTEM

Brugervejledning

af

Bent Bæk Jensen
Ole Lehrmann Madsen
Bent Brun Kristensen
Søren Henrik Eriksen

Marts 1973

INDHOLD

INDLEDNING

BOBS-system er det første led i en compiler-compiler, forstået således at givet en grammatik producerer BOBS-system en syntax-checker for det pågældende sprog defineret ved grammatikken. Syntax-checkeren leveres i form af et Pascal-program, som desforuden indeholder en leksikalsk analyse og en error-recovery.

Program-komplekset består af to hoveddele:

1. Parser-generatoren
2. Parseren

Nærmere beskrevet er parser-generatoren en såkaldt SLR(1)-parser-generator (simple left-right, one look-ahead), hvor SLR(1) er en type af sprog genereret af en SLR(1)-grammatik. Disse sprog kan placeres i den sædvanlige Chomsky klassification: SLR(1) ligger imellem Ch. type 3 og Ch. type 2, således at precedence-grammatikker er inkluderet i SLR(1)-grammatikker. For yderligere at give et indtryk af hvor omfattende SLR(1) er, kan det nævnes, at ALGOL 60 med få modifikationer er SLR(1).

SLR(1)-grammatikkerne indgår som en ægte delmængde af LR(k)-grammatikkerne, der er defineret på følgende måde:
Et kontext frit sprog er LR(k), hvis og kun hvis der gælder, at man ved et enkelt deterministisk scan fra venstre (L-left) til højre (R-right) ved højst at kigge k-symboler frem på input-strengen (k look-ahead) kan parse strenge i det pågældende sprog. Det væsentligste i denne sammenhæng er look-ahead mængderne. I vores tilfælde (SLR(1)) skal vi kun kigge eet symbol frem på input. Den præcise definition af look-aheadmængden (LA) for et symbol i V = VT ∪ VN er :

for $A \in VT$ er $LA(A) = \{A\}$
for $A \in VN$ er $LA(A) = \{(1:\beta) \in VT \mid s \rightarrow^* \alpha A \beta$ hvor $\alpha, \beta \in V^*\}$

hvor der med $(1:\beta)$ menes det første symbol i strengen β.

Forklaret i ord er look-ahead mængden for en non-terminal de terminal-symboler, der kan følge umiddelbart efter denne.

Det man forsøger at opbygge som parser, er en DPDA (dvs. en deterministisk push-down automat), der på grundlag af stakken og input-symbolet afgør om man skal reducere eller læse videre på input. Er der tvivl, bruger man så look-ahead-mængderne til at afgøre, hvad man skal foretage sig.

En grammatik er SLR(1), hvis disse look-ahead-mængder inden for samme tilstand i parseren er disjunkte.

Et eksempel:

Betragt grammatikken

$VN = \{E, T, P\}$

$VT = \{+, *, (,), I\}$

1) $E \rightarrow E + T$

2) $E \rightarrow T$

3) $T \rightarrow P * T$

4) $T \rightarrow P$

5) $P \rightarrow (E)$

6) $P \rightarrow I$

Lad os finde look-ahead mængden for non-terminalen P. P forekommer på to højresider, i produktion nr. 3) og 4). Af 3) får vi, at $* \in LA(P)$. 4) implicerer, at $LA(T) \subseteq LA(P)$, dvs, at vi nu skal finde LA(T). T forekommer på højreside i produktion 1), 2) og 3). Vi ser, at både 1) og 2) giver, at $LA(E) \subseteq LA(T)$. 3) giver $LA(T) \subseteq LA(T)$ (som er uinteressant). Vi skal altså finde LA(E) . E forekommer på højreside i produktion 1) og 5). Af 1) får vi, at $+ \in LA(E)$ og af 5) får vi, at $) \in LA(E)$.
Ialt fås, at: $LA(P) = \{*, +,)\}$.

PARSERGENERATOREN

Parsergeneratoren er opbygget i tre hoveddele:

1. Indlæsning
2. Grammatik-check
3. Opbygning af syntax-checkeren

Her vil kun de to første punkter blive behandlet, da punkt 3 kræver et indgående kendskab til DeRemer's metoder til opbygning af SLR(1) parsere; altsammen beskrevet i [1].

1. Indlæsning

De følgende sider er en nærmere definition af, hvorledes grammatikken skal se ud for at være syntaktisk korrekt.

Input skal bestå af følgende ting i den angivne rækkefølge.

1. Indlæsning af metasymboler
2. - - terminalsymboler
2a. - - string beskrivelse
2b. - - goalsymbol
3. - - grammatik I (BNF)
3a. Evt. skift af metasymboler.

Det følgende er en uddybning af disse seks punkter:

Pkt. 1:

Metasymbolerne specificeres på et kort med føl gende udseende:

---METASYMBOLS--M1=TEGN1---M2=TEGN2--M3=TEGN3--M4=TEGN4--

Alle steder, hvor der er markeret et space (-), må der være nul eller flere spacer. Alle andre steder accepteres space ikke. De fire metasymboler skal vælges som fire forskellige enkeltkarakterer.

Betydningen af M1-M4 er som følger:

Tegn1-Tegn4 skal være enkeltsymboler (se iøvrigt under 3a), men dog ikke space og eol (end-of-line), og de svarer til følgende tegn i normal BNF:

Tegn1 (M1) svarer til ::=

Tegn2 (M2) svarer til /

Tegn3 (M3) svarer til < og >

Tegn3 må ikke være M eller G, da dette i specielle tilfælde kan give fejl.

Tegn4 (M4) svarer til afslutningen på en produktionskæde med samme venstreside (se under pkt. 3a).

Metasymbolkortet kan udelades, hvis man ikke har specielle ønsker med hensyn til valg af metasymboler. Som default-værdier fås M1==, M2=/, M3=< og M4=; .

Eksempler:

METASYMBOLS M1== M2=/ M3=< M4=;

Dette er et eksempel på et rigtigt skrevet metasymbolkort (i det følgende vil disse metasymboler blive anvendt i eksemplerne hvis ikke andet er nævnt).
Følgende vil derimod ikke være rigtigt og vil give anledning til forskellige fejludskrifter.

META SYMBOLS o. s. v.

eller

METASYMBOLS M1== M 2=/ o. s. v.

eller

METASYMBOLS M1 =1 M2=/ M3= < < o. s. v.

Det sidste p. g. a. < <

Pkt. 2 :

Karaktersættet deles op i to klasser:

1. Bogstaver og cifre
2. Alt andet undtagen space og eol

Terminaler skal bestå af en eller flere karakterer fra enten den ene eller den anden af disse to grupper, men ikke dem begge. Dvs. at et terminalsymbol først er afsluttet, når man møder et space, eol eller en karakter fra den anden mængde.
Space og eol vil altid blive opfattet som delimiter for terminalsymboler. Denne opdeling bevirker også, at to terminaler fra samme klasse, som står ved siden af hinanden uden delimiter, vil blive opfattet som en sammensat terminal.

Eksempel:

beginend opfattes som beginend og ikke som begin og end.

Eksempel på ulovligt sammensatte terminaler:

. ne. ≠begin≠ -e-n-d

Eksempel på lovlige terminaler:

begin end := = (+ if then

Terminalerne skrives på et eller flere kort med mindst et space imellem. Et sammensat terminalsymbol (f. eks. begin) må ikke deles over flere kort. Efter sidste terminal skal tegn4(M4) stå.

Terminaler på over ti karakterer accepteres ikke, kommer der flere karakterer giver det anledning til en fejlmeddelelse.

Der eksisterer en parserversion, der på lexikalsk niveau opsamler navne, konstanter og strenge. Denne opsamling opnås ved blandt terminalerne at specificere følgende terminalsymboler NAME, KONST eller STRING. (Se iøvrigt under afsnittet om navne, konstanter og strenge.)
Hvis faciliteten med opsamling af navne, konstanter og strenge benyttes, må ingen terminal begynde med et ciffer.

Den tomme streng specificeres som terminalen EMPTY. Dvs. hvis man ønsker at anvende nonterminaler, der kan derivere den tomme streng, skal man under indlæsningen af terminalerne huske at have EMPTY med. Den sammensatte terminal vil, hvis den er specificeret,under alle omstændigheder blive opfattet som den tomme streng (dog må EMPTY gerne bruges som nonterminal). Anvendes EMPTY, skal den være eneste symbol på højresiden.

Eksempel:

Gyldig	<AA< = EMPTY / B / C <BB< ;
Ugyldig	<AA< = <BB< EMPTY <CC< ;

Pkt. 2a:
Ønsker man at anvende de specielle faciliteter vedrørende NAME, KONST og STRING (se nærmere under beskrivelsen af parseren), kræver sidstnævnte en specification af en string-escape-karakter for strenge. Til dette formål skrives et stringch-kort af følgende form:

--STRINGCH = --<enkelt-karakter>--tegn4

bemærk, at den specificerede escape-karakter ikke må benyttes i anden sammenhæng i grammatikken.
I sproget Pascal har man stringch = ≡ , hvilket kunne specificeres på flg. vis:

STRINGCH = ≡ ;

Angående anvendelse henvises til afsnit under parseren.

Pkt. 2b:
Da udvælgelsen af et goalsymbol er essentiel for hele parserideen, skal et sådant specificeres.
Der er følgende to muligheder:

1. at skrive et kort som specificerer et goalsymbol
2. at lade første venstreside, der optræder, være goalsymbol.

1:
Et kort med følgende udseende klarer det fornødne (m.h.t. spacer se under pkt. 1):

---GOALSYMBOL=--nonterminal--tegn4

Her skal nonterminal være en nonterminal, som forekommer i grammatikken, og man skal huske at afslutte kortet med metasymbol 4.
Hvis dette kort anvendes skal det placeres umiddelbart efter det sidste kort med terminalsymboler eller efter stringch-kortet (se under pkt. 2a.).

2:
Ønsker man,at den første venstreside skal optræde som goalsymbol, undlader man blot kortet beskrevet under pkt. 1.

Eksempler:

METASYMBOLS M1== o.s.v.

{ terminaler

GOALSYMBOL = PROGRAM ;

{ grammatik

Dette vil have samme virkning som følgende:

METASYMBOLS M1== o. s. v.

{ terminaler

<PROGRAM< = BEGIN <BLOCK< END ;
<BLOCK< = ... o. s. v.

{ resten af grammatikken ;

Pkt. 3:
Grammatikken skrives i den modificerede BNF (beskrevet under pkt. 1) med de metasymboler, som er specificeret på metasymbolkortet. Efter sidste produktion med samme venstreside skrives metasymbol 4. Dette skal dog ikke forstås sådan at produktioner med ens venstreside ikke må skrives hver for sig.

Eksempel:

<D< = A / B /<DD< B/ E ;

vil blive opfattet på samme måde som

<D< = A/B;
<D< =<DD<B/ E ;

Endelig skal der efter den sidste produktion i grammatikken ikke skrives et men to tegn4 (M4).

Eksempel på en rigtigt skrevet grammatik:

<PROGRAM<= <EXPRESSION<;
<EXPRESSION< = <EXPRESSION< + <TERM< / <TERM< ;
<TERM<= <PRIMARY< * <TERM< / <PRIMARY< ;
<PRIMARY< = (<EXPRESSION<)/ I ; ;

Nonterminaler må bestå af samtlige karakterer undtaget den karakter, der anvendes som metasymbol 3. Det skal bemærkes, at non-terminaler på mere end 30 karakterer ikke accepteres og at alle steder under indlæsningen er space og eol blinde (dette gælder dog kun non-terminaler (m.h.t. terminaler se under pkt. 2)).

Eksempel:

```
<PROG   R AM  < ={   <EXP  <   }   ;
```

Dette opfattes som

```
<PROGRAM<={<EXP<};
```

Pkt. 3a:

Skift af metasymboler:

Da BNF'en her kræver fire metasymboler, bliver karaktersættet reduceret med fire symboler, da de fire anvendte tegn ikke må indgå som dele af hverken terminaler eller non-terminaler. Ønsker man at anvende et eller flere af metasymbolerne ved skrivning af grammatikken, har man mulighed herfor ved at skifte metasymboler midt under indlæsningen af grammatikken. Dette gøres ved at skrive et kort som under pkt. 1, hvor man uanset om man skifter et eller flere symboler, skal skrive hele kortet som specificeret under pkt. 1. Ved et metaskift frigøres de hidtil anvendte metasymboler og disse kan nu frit bruges.

Eksempel:

```
METASYMBOLS M1 ==  M2=/  M3=<  M4=;

{ terminal definition

<PROGRAM<= BEGIN<STATEMENT< END;
<STATEMENT<=<EXP</ EMPTY;
METASYMBOLS M1 ==    M2=M  M3=5    M4=;
5 EXP 5 =IF  5 B005  THEN 5EXP5 M EMPTY;
```

Indlæsningen forsøger så vidt muligt at hjælpe brugeren til at få skrevet sin grammatik syntaktisk korrekt. Hvis der opdages en fejl i input, udskrives en relevant fejlmeddelelse, og ved mindre alvorlige fejl skippes til nærmeste metasymbol 4, hvor indlæsningen genoptages. På denne måde skulle antallet af kørsler gerne blive reduceret væsentligt, idet man hver gang får så mange fejl ud som muligt, før den videre kørsel standses.

2. Grammatik-check

Efter at grammatikken er blevet godkendt som syntaktisk korrekt ved indlæsningen, checkes det, om grammatikken opfylder visse krav:

a) Venstre- og højre-rekursivitet.

Det undersøges om nogle non-terminaler er venstre- og/eller højre-rekursive. Hvis "både og" er grammatikken tvetydig (ambiguous) og dette bevirker, at programmet standser, da man ikke kan opbygge parseren for tvetydige grammatikker.

b) Termination.

Det undersøges, om alle non-terminaler kan derivere en streng af lutter terminal symboler, hvis ikke, standser programmet.

c) Erasure.

Det undersøges, om nogen non-terminal kan derivere den tomme streng. Hvis det er tilfældet, modificeres grammatikken, og programmet fortsætter. I denne forbindelse skal det nævnes, at under indlæsningen checkes det, om den tomme streng (EMPTY) er skrevet i rigtig sammenhæng ifølge input definitionen.

d) Identiske produktioner.

Det undersøges, om der er ens produktioner i grammatikken. Hvis der er, fjernes dubletterne.

e) Ubrugte non-terminaler.

Det undersøges, om alle non-terminaler (pånær goalsymbolet) forekommer som både venstre og højre side i en produktion. Hvis ikke, standses den videre kørsel.

f) Fjernelse af kæder.

Det undersøges, om der forekommer en produktionskæde af flg. form:

nonterm I → nonterm I+1

Eksisterer sådanne, fjernes disse.

Eksempel:

$$X_j \rightarrow \varphi_j \; A \; \psi_j \qquad j = 1, 2, \ldots, k$$

$$A \rightarrow B$$

$$B \rightarrow C$$

$$C \rightarrow I \text{ (I terminal)}$$

vil reduceres til

$$X_j \rightarrow \varphi_j \; C \; \psi_j \qquad j = 1, 2, \ldots, k$$

$$C \rightarrow I$$

g) Sammenhæng.

Det undersøges, om alle non-terminaler kan deriveres fra goal-symbolet.

Output fra parsergeneratoren

Output fra parsergeneratoren er delt op i fem afdelinger:

1. Listning af input
2. Grammatik check
3. Endelig grammatik
4. SLR(1) output
5. Error message

1. Listning af input

Her får man en direkte kopi af det læste input eventuelt med fejlmeddelelser.

2. Grammatik check

Viser hvilke grammatik check der er foretaget, samt i hvilken rækkefølge, det er sket. Desuden bliver der skrevet ud, hvilke modifikationer der er foretaget.

3. Endelig grammatik

Er den grammatik, som bruges til opbygning af parseren. Den er dannet ud fra den indlæste grammatik ved de ændringer, der er foretaget under grammatik checkene. Listningen er skrevet i BNF med de tilhørende produktionsnumre skrevet til venstre.

4. SLR(1) output

a) er den endelige grammatik SLR(1), bliver der på output skrevet: the grammar is SLR(1).

b) non SLR(1) states

er den endelige grammatik ikke SLR(1), er det fordi der findes inadequate tilstande, det vil sige, at mængden af terminalsymboler og look-ahead-mængderne for de enkelte reduktioner ikke er parvis disjunkte.

For hver tilstand kan der forekomme to typer af udskrifter.
Den første er en listning af mængden af terminalsymboler med oplysning om hvor de optræder i grammatikken (produktionsnummer samt terminalsymbolets nummer i produktionens højre side). Denne mængde kan være tom. Den anden type er en listning af look-ahead-

mængderne for hver indgående reduktion med oplysning om produktionens nummer. (M.h.t. nærmere oplysninger om inadequate tilstande henvises til [1] eller [2].)

Eksempel:

```
**********************  A LIST OF NON-SLR1 STATES  **********************

        IN EACH OF THE FOLLOWING STATES THE
        SETS OF SYMBOLS ARE NOT DISJOINT

        ------------------------------------------------------------------

        READING     CONTINUES IF THE NEXT INPUTSYMBOL
                    IS ONE OF THE FOLLOWING :
                    SYMBOL        PRODUCTION     SYMBOL NO
                     EXP                 5           1

        REDUCTION   NO 100 IS PERFORMED IF THE NEXT INPUTSYMBOL
                    IS ONE OF THE FOLLOWING :
                    D
                    C

        REDUCTION   NO 73 IS PERFORMED IF THE NEXT INPUTSYMBOL
                    IS ONE OF THE FOLLOWING :
                    IF
                    D

**********************  END OF NON-SLR1 STATES  **********************

**************************************************************************

        THE GRAMMAR IS NOT SLR1
        THE PROGRAM STOPS

**************************************************************************
```

c) warning message

Ved visse uheldige kombinationer af tilstande og produktioner kan den interne værdi af NAME (se nærmere i afsnittet om navne, konstanter og strenge) mistes. Problemet opstår, hvis der i en af look-ahead-mængderne for en reduktion i en inadequat tilstand forekommer NAME, og NAME også samtidig forekommer på produktionens højre side. Den interne værdi af det genkendte NAME vil blive overskrevet dersom næste input ord er et NAME.
Det samme gælder for KONST og STRING.
I warning message listes produktionsnummer, tilstandsnummer samt hvilke værdier (NAME, KONST eller STRING), man kan miste.
Det tilrådes, at man laver grammatikken im, hvis der forekommer et warning message.

Eksempel:

```
*****************************  WARNING MESSAGE  *****************************

        THE INTERNAL VALUE OF KONST,NAME OR STRING
        MAY BE LOST IN A COMBINATION OF THE
        FOLLOWING PRODUCTION(S) AND STATE(S)

 PRODUCTION NO.    STATE NO.   INTERNAL VALUE OF
       6                3            STRING KONST
       4                5            NAME

*****************************************************************************
```

5. Error message table

Er den endelige grammatik SLR(1) produceres en error message table til brug ved fortolkning af output fra den producerede parser. Parseren markerer fejl i input-strengen med sammenhørende pile og fejlnumre. Tabellen angiver nu sammenhæng mellem fejl -nummer og mængden af terminalsymboler, der kunne forventes i stedet for fejl-symbolet.

PARSEREN

Parseren er som før nævnt et Pascal-program, hvori parsergeneratoren initialiserer tabeller og konstanter. Foruden grammatikken skal generatoren derfor have selve parseren som input. Dette foregår i praksis ved at denne i forvejen skal ligge på den lokale file 'PARSIN' Det færdige resultat leveres så på den lokale file 'PARSOUT'. Parseren er en afsluttet enhed, der uden ændringer kan checke syntaksen for en input-streng. Ønsker man yderligere at tilføje semantiske aktioner, er der indbygget behjælpelige elementer, der er beskrevet i det følgende.

Parseren indeholder i det væsentlige følgende:

1. Leksikalsk analyse
   a) standardversion
   b) udvidet version med opsamling af navne, konstanter og strenge.
2. Syntaks analyse
3. Error-recovery

Leksikalsk analyse

Da effektiviteten af den leksikalske analyse er et springende punkt, findes der foreløbig to versioner af parseren med hver sin leksikalske analyse. Den ene af disse parsere indeholder en generel, men simpel leksikalsk analyse. Den anden parser indeholder en leksikalsk analyse, som opsamler navne, konstanter og strenge. Anvender man traditionelle navne, konstanter og strenge i sit sprog, vil det ofte være fordelagtigt at benytte den sidstnævnte parser.

Som nævnt under indlæsningens punkt 2 er der indført restriktioner på, hvordan de sammensatte terminaler må se ud. Disse restriktioner er indført, fordi man vil undgå backtracking og lignende tidskrævende manøvrer.

Følgende konstruerede eksempel viser klart hvorfor. Antag at vi har følgende terminaler: BEGIN, END, BE, GIN, ENDE, BEGINEND og ELSE. Hvordan skal nu input-strengen BEGINENDELSE opfattes? Reglerne om sammenstilling af terminalsymboler kan af og til virke ufleksible. Derfor er der indført visse undtagelser for at gøre tilværelsen behageligere. Men det skal kraftigt pointeres at disse undtagelser kun gælder for parseren. Undtagelserne beskrives nærmere i afsnittet om de enkelte parsere.

Afsnit A

For standardversionen gælder følgende:

1) Terminalsymboler må sammenstilles som beskrevet under indlæsningens punkt 2.
2) Mødes en sekvens af karakterer som ikke tilsammen udgør et sammensat terminalsymbol, vil hver enkelt karakter blive opfattet som et terminalsymbol.

Eksempel:

Betragt følgende grammatik:

<S< = A <S< A / B <S< B / C ;

Strengen ABC BA vil blive opfattet som

A B C B A

Afsnit B

Følgende eksempler viser, hvordan sammenstilling af terminalsymboler kan virke ufleksibel i de programmeringssprog, vi normalt arbejder med.

Eksempel:

Terminalsymboler : := X Y 3 4 + - ( ) / ;

Ifølge reglerne må man ikke skrive:

X:=(Y+((X-3)/4));

men skal nødvendigvis skrive:

X:= (Y+ ((X-3) /4)) ;

For at undgå disse restriktioner har man for klasse tos vedkommende indført at 'visse' terminaler godt må stå ved siden af hinanden, hvis følgende regler overholdes:

1. Terminalerne (enkeltsymboler og sammensatte) må ikke tilsammen udgøre et af de andre symboler.
2. Terminalerne må ikke udgøre begyndelsen på et af de andre terminaler.

Under punkt 1 gælder, at terminalerne opfattes som det sammensatte terminal, de tilsammen udgør.

Eksempel:

Terminaler := : = X Y
Strengen X:=Y opfattes som X := Y
og ikke som X : = Y

Under punkt 2 gælder, at terminalerne vil blive markeret som en fejlsammensætning. Begrundelsen herfor er, at der altid ledes efter den længste forekomst af en terminal, og at man ikke vil backspace på input og lave backtracking m.m.

Eksempel:

Terminaler := : = ::== (
Strengen ::=(opfattes som ::= (
hvor ::= er en fejlagtig terminal
og ikke som : : = (eller : := (

I tvivlstilfælde kan man altid selv indsætte ekstra spacer, så man får strengen genkendt på den ønskede måde.

Navne, konstanter og strenge

I de fleste højere programmeringssprog idag opererer man med navne, talkonstanter og karakterstrenge. Disse erklæres ofte på følgende måde:

<identifier> ::= <letter> / <identifier> <letter>
/ < identifier> <digit>

<digit> ::= 0/1/../9
<letter> ::= A/B/C/...../Z
<integer> ::= <integer> <digit> /<digit>
<string> ::= ≡ <any sequence of characters not containing ≡ > ≡

Da opsamlingen af disse ting er meget plads- og tidskrævende for syntax-delen, foretages denne derfor i den leksikalske analyse. Blandt terminalerne har vi derfor indført følgende specialsymboler:

1) NAME (som står for <identifier>)
2) KONST (som står for <integer> m.m.)
3) STRING (som står for <string>)

Hvis parseren møder en streng af symboler fra klasse 1, og det gælder, at strengen ikke er defineret som en terminal, så opfattes strengen som terminalen NAME eller som terminalen KONST. Strengen opfattes som NAME, hvis den begynder med et bogstav, som KONST, hvis den begynder med et ciffer.

Anvendes string faciliteten, vil parseren opfatte en karakterstreng mellem to string-escape-karakterer som terminalen string. (Bemærk at escape-karakteren skal betragtes som en terminal.)
Ønsker man, at en escape-karakter skal være element i selve strengen, angives dette ved at duplikere den.

Eksempler: (STRINGCH= ≡)

Flg. strenge opfattes som name: FIRST B117 PL1
– – – – konst: 17 11B 318 1B2M3ABC
– – – – string: ≡BOBS≡ ≡+,-,^≡ ≡≡≡A≡≡≡
(≡≡≡A≡≡≡≡B≡≡≡ angiver string ≡A≡≡B≡)

Alle steder i grammatikken, hvor <identifier> forekommer, kan man skrive symbolet NAME. Alle steder, hvor <integer> forekommer, kan man skrive KONST, og endelig kan man skrive STRING på lignende relevante steder.

Hvis man ønsker, at de ovennævnte ting skal opsamles i den leksikalske analyse, skal NAME, KONST og STRING erklæres som terminaler.

Eksempel:

```
Terminaler + ( ) * NAME KONST STRING ;
STRINGCH= ≡ ;
<E< =<E< + <T< / <T< ;
<T< = <P< * <T< /<T<;
<P< = NAME / KONST / STRING /(<E<);;
```

Input streng til parseren:

≡TOLV≡+(LAST* 81+17B)+NEW

Dette opfattes som

STRING + (NAME * KONST + KONST) + NAME

Under parsningen er det selvfølgelig ikke nok at vide, at man lige har parset et navn, konstant eller string, men også hvilket navn det var. Analogt for konst og string.
Der er derfor indført følgende:

1. Name

   Parseren gemmer de læste navne i en tabel sammen med en talværdi, der entydigt bestemmer navnet. Endvidere eksisterer en global variabel, som hedder NAME, som altid indeholder talværdien for det sidst læste navn. Når f. eks. reduktionen p → NAME udføres, må man selv i proceduren code (se under syntaxanalyse) indføre en kode, der gemmer navnet af vejen.

2. Konstanter

   I arrayet KONST opbevares den læste konstant, som en karakterstreng fra celle 1 til den celle KONSTNO peger på. Man må så selv afgøre, hvordan strengen skal opfattes.

3. String

I arrayet STRING opbevares den læste karakterstreng fra celle 1 til og med celle STRINGNO.
(Bemærk at de omgivende escape-karakterer vil være skippet.)

Bemærk, at NAME ikke bør forekomme to eller flere gange på samme højreside. Endvidere bør en nonterminal til højre for NAME i én regel ikke kunne derivere NAME. Det samme gælder for KONST og STRING. Under konstruktion af grammatikken bør man være opmærksom på dette.

Eksempel:

Betragt følgende grammatik

(1) <type> ::= NAME=(<namelist>);
(2) <namelist> ::= NAME
(3) / <namelist> ,NAME

Antag nu vi har flg. inputstreng:
FARVE=(RØD, GRØN, GUL)

Når reduktionen (1) udføres, vil variablen name ikke indeholde internværdi af farve, men den interne værdi af gul, da gul er det sidst læste navn. Da det som regel ikke er det man ønsker, kan man omskrive grammatikken til flg.:

(1) <type> ::= <typename>=(<namelist>)
(2) <namelist> := NAME
(3) / <namelist> , NAME
(4) <typename> := NAME

Denne grammatik giver mulighed for at gemme de rette NAMES væk på de rigtige tidspunkter.

Generelt kan man for at undgå konflikter af ovennævnte art indføre følgende produktioner:

<name> ::= NAME
<konst> ::= KONST
<string> ::= STRING

2. Syntaxanalyse

Syntaxanalysen anvender de af parsergeneratoren producerede tabeller til at parse inputstrengen. Når en højreside i en regel er genkendt, kaldes proceduren CODE (produktionsnummer). De produktionsnumre CODE kaldes med,er de numre, der står i grammatikudskriften fra parsergeneratoren. Meningen er så, at man i en case-konstruktion i CODE kan foretage en til produktionen relevant semantisk aktion.

3. Error recovery

Error recovery forsøges, når syntax-checkeren opdager en fejl, dvs. næste symbol på input ikke kan efterfølge den allerede parsede tekst.

Fejlen markeres med en pil under fejlsymbolet og med et tilhørende nummer i højre margin til brug ved opslag i tabellen "compiler error messages" .

En * yderst i højre margin markerer mere end 10 fejl i linien, og at kun de 10 første er markeret.

Den anvendte metode til udbedring af fejl er kort skitseret flg.:

1. Lokal fejl-søgning

   Efter tur prøves en af følgende muligheder:

   a. Indsættelse af ekstra relevant symbol før fejlsymbolet.
   b. Udeladelse af fejlsymbolet.
   c. Indsættelse af relevant symbol i stedet for fejlsymbolet.

Er genoptagelse af parsningen stadig ikke mulig forsøges:

2. Global error recovery

   Skip symboler på input indtil man møder et symbol, ved hjælp af hvilket man sammen med den information, der er i stakken kan genoptage parsningen på rimelig vis.

Output fra parseren.

Foruden listning af input med relevante fejlmeddelelser vil der i jobbets dayfile gives information om parsningens udfald.

APPENDIX A

Citat fra David Gries:

> Compiler construction for digital computers.
> "Don't take the debugging facilities out of the compiler when you think the compiler is debugged. It definitely isn't."

Options.

Under indkøringen af parsergeneratoren har vi undervejs indført en del testudskrifter med mere.
Da nogle måske kan få glæde af disse, skal her kort skitseres, hvordan de bruges.
Hvert test (eller option) har fået et nummer, som det kaldes med. Hvis man ønsker at aktivere nogle af de beskrevne options skrives som første inputstatement til generatoren følgende:

OPTIONS (<numrene på de ønskede options>)

Eksempel:
OPTIONS (2,10,3,20,25)
METASYMBOLS M1==M2=/ M3=< M4=↓
osv.

Der gøres opmærksom på, at BOBS ikke føler sig ansvarlig, hvis brugen af options giver anledning til fejl.

N.B. Nogle options kan levere enorme mængder af output, hvis grammatikken er stor. Derfor bør man anvende disse testudskrifter med varsomhed.

Liste over benyttede options

1: Intern værdi af samtlige terminaler udskrives.
2: LR0-maskinen udskrives på intern form.
3: Samme interne værdi til flere terminalsymboler (kun sidstnævnte vil optræde i udskrifter.)

Eksempel:

METASYMBOLS M1 == M2= osv.

DELETE DEL = D = SAVE S = osv.

Her vil delete, del og d få samme interne værdi ligesom det samme vil gælde for save og s. Effekten opnås ved f.eks. efter del og d at skrive en space efterfulgt af m1 efterfulgt af en space.

4: Terminalsymboler på mere end 10 karakterer vil blive accepteret således,at alle karakterer er signifikante, men kun de 10 første vil optræde i udskrifter.

5: Fri mulighed for at indføre terminalsymboler bestående af karakterer fra begge char-mængder. Under indlæsning af grammatikken vil nu kun space, eol og m1 til m4 være skillende tegn.
N.B.! Ved benyttelse af denne option må man selv omskrive leksikalsk analyse i parseren.

6: Intern værdi af samtlige nonterminaler udskrives.

7: Look-ahead-mængden for samtlige nonterminaler udskrives.

8*: Ingen listning af input.

9*: Ingen output fra grammatikcheck.

10*: Ingen udskrivning af den endelige grammatik.

11: Ingen udskrivning af "error message table".

14: Udskrivning af den reducerede parser (DPDA) på intern form (efter pres).

15: Udskrivning af LR0-maskinen med lookbacktilstande (intern form efter lookback).

16: Udskrivning af array "PROD" (efter lookback).

17: Udskrivning af array "TILSTAND" (efter lookback).

18: Udskrivning af terminale ethoveder og ethaler for nonterminaler.

19: Test af "VHRECURS" (tre lokale variable udskrives).

20: Test af "VHRECURS" (bitmatricer udskrives).

* Hvis option 8, 9 eller 10 er sat, leveres ikke altid output fra de andre options.

APPENDIX B

Det følgende er en liste af de fejludskrifter, der kan komme fra BOBS-system.

Fejl vedrørende indlæsningen:

EOF ENCOUNTERED IN READING THE GRAMMAR

Under indlæsningen af grammatikken mangler der enten to metasymbol 4 efter sidste produktionskæde eller der er fejl andetsteds.

METASYMBOL 3 APPEARS IN WRONG CONTEXT

Denne fejl kan komme uden metasymbol 3 er involveret, men fejlen er da af alvorligere karakter. F.eks. kan man have glemt metasymbol 4 efter en produktion eller lignende.

INPUT FILE EMPTY

Programmet har ikke kunnet finde noget input.

= EXPECTED IN GOALSYMBOLCARD

GOALSYMBOLCARD EXPECTED

EMPTY IS NOT THE ONLY SYMBOL IN RIGHTSIDE

EMPTY er brugt forkert. Se pkt. 2 under indlæsningen.

ERROR(S) IN METASYMBOLS
METASYMBOLCARD EXPECTED
MORE THAN ONE METASYMBOL2 HAS BEEN MET
ERROR IN RIGHTSIDE, METASYMBOL2 FOLLOWING METASYMBOL1
TWO OR MORE METASYMBOLS ARE ALIKE
METASYMBOL1 WAS EXPECTED
END OF FILE ENCOUNTERED.
TERMINAL DUPLICATED.

Et terminalsymbol er erklæret mere end en gang.

(Fejlen er ikke fatal).

TERMINAL MORE THAN 10 CHARACTERS.

ILLEGAL CONCATENATION.
En terminal består af karakterer fra både klasse 1 og klasse 2 (se indlæsningen pkt. 2)

ILLEGAL TERMINAL.

ILLEGAL CHARACTER IN OPTION LIST
Dette er ikke en fatal fejl.

OPTIONNUMBER IS NOT DEFINED
Der er specificeret et optionnummer større end den største optionværdi (konst 18). Fejlen er ikke fatal.

IT WAS THE MOST SICK PARSIN FILE I HAVE EVER SEEN
PLEASE TRY AGAIN WITH ANOTHER ONE
Filen PARSIN indeholdt ikke den korrekte parser.

ERROR(S) IN GRAMMAR
Grammatikken indeholder visse logiske fejl. Se meddelelser i outputdelen under grammatikcheck.

De følgende fejludskrifter har noget at gøre med de konstanter, der er erklæret i begyndelsen af parsergeneratoren.

TOO MANY NONTERMINALS -CONST2 TOO SMALL

CLASS NONTERM EXCEEDED -CONST1 TOO SMALL

-ERROR IN LOOKAHEAD. CONST8 TOO SMALL

NUMBERS IN TERMTREE EXCEEDING MAXIMUM. (CONST12).

NUMBER OF TERMINALS EXCEEDING MAXIMUM. (CONST2).
ERROR -CLASS RIGHTSIDE TOO SMALL(CONST4)

ERROR IN INITPROD. CONST1 TOO SMALL

ERROR IN LOOKB. CONST10 TOO SMALL

ERROR IN VHRECURS
CONST13 TOO SMALL
IT MUST BE NN

CLASS PARSER TOO SMALL
CONST6 TOO SMALL

TOO MANY PRODUCTIONS FOR A SINGLE STATE
CONST14 TOO SMALL
THE PROGRAM STOPS

TOO MANY WARNING STATES
CONST14 TOO SMALL
THE PROGRAM STOPS

De følgende fejludskrifter begynder alle med ERROR IN LR(0).

MAXIMUM NUMBER OF ELEMENTS IN CLASS RIGHTSIDE EXCEEDED (CONST4).

UPPER LIMIT OF CONST9 REACHED.

UPPER LIMIT OF CONST5 REACHED.

MAXIMUM NUMBER OF ELEMENTS IN CLASS PARSER EXCEEDED (CONST6).

UPPER LIMIT OF CONST10 REACHED.

APPENDIX C

Det følgende er et eksempel på anvendelse af BOBS systemet, samt en beskrivelse af de permanente files, der udgør systemet.
Den permanente file:

BOBSSYSTEM, ID=DATZZ

indeholder systemet, der udgøres af følgende 5 cycler, alle katalogiseret med multiread.

cy=5.	Binær version af generatoren
cy=4.	Textversion af generatoren
cy=3.	Textversion af parseren (med name,konst og string)
cy=2.	Textversion af parseren (uden name,konst og string)
cy=1.	Nyeste version brugervejledning (denne cycle indeholder forrest et program, der udskriver teksten på pæn form. Fra en terminal kan man så f. eks. gøre følgende: ATTACH, PP, BOBSSYSTEM, ID=DATZZ, CY=1. BATCH, PP, INPUT.)

Den binære version på cy=5 kræver en fieldlength på 50000 (octalt). Hvis ens grammatik er stor, vil konstanterne på denne cycle være for små, og man kan henvende sig til et medlem fra BOBS gruppen og få genereret en version med større konstanter.

Det er selvfølgelig ikke givet, at de nævnte cycles vil eksistere til tid og evighed.

Som tidligere nævnt findes en FORTRAN version af parseren. Ønsker man at benytte denne, kan man henvende sig til et medlem af BOBS gruppen.

Eksempel

De næste sider er et eksempel på anvendelse af systemet. Først en listning af input til generatoren med de nødvendige styrekort:

```
DATZZ,CM47000,T50.
ATTACH,BIN,BOBSSYSTEM,ID=DATZZ,CY=5.
ATTACH,PARSIN,BOBSSYSTEM,ID=DATZZ,CY=3.
COMMENT.
COMMENT.
PASCAL,LOAD=BIN,LL=2000.
CATALOG,PARSOUT,PARSER,ID=DATZZ,CY=1.
```

Den binære version af generatoren, parser-versionen med name, konst og string (en del af input til generatoren)

```
METASYMBOLS M1=# M2=/ M3=< M4=!
 BEGIN  END REPEAT WHILE DO UNTIL IF THEN ELSE
KONST VAR  NAME  INTEGER ( )
+ * := , . ; !
  <PROGRAM< # <VARIABLE PART<   <STATEMENT SEQUENCE< END . !
 <VARIABLEPART< # VAR <DECLARATIONLIST< !
 <DECLARATIONLIST< # <DECLARATIONLIST< ;  <DECLARATION <
                   / <DECLARATION< !
<DECLARATION< # INTEGER <NAMELIST< !
<NAMELIST< # <NAMELIST< , NAME / NAME !
<EXPRESSION< # <EXPRESSION< + <TERM < / <TERM< !
<TERM< # <PRIMARY< * <TERM< / <PRIMARY< !
<PRIMARY< # ( <EXPRESSION< ) / NAME / KONST  !
<IFCLAUSE< # IF <EXPRESSION< THEN !
<TRUEPART< # <IFCLAUSE< <CLOSED STATEMENT< ELSE !
 <WHILE< # WHILE !
 <WHILECLAUSE< # <WHILE<  <EXPRESSION< DO !
<REPEAT< # REPEAT !
<STATEMENT SEQUENCE<# BEGIN /  <STATEMENT SEQUENCE<
                    < STATEMENT< ; !
<CLOSED STATEMENT< # <REFERENCE< :=<EXPRESSION<
         / <STATEMENT SEQUENCE< END !
<REFERENCE< # NAME !
 <STATEMENT*< # <CLOSED STATEMENT<
          / <TRUEPART< <STATEMENT*<
         / <IF CLAUSE< <CLOSED STATEMENT<
         / <WHILE CLAUSE< <STATEMENT*<
         / <REPEAT< <STATEMENT< UNTIL <EXPRESSION< !
<STATEMENT< # <STATEMENT*< ! !
```

```
************ A LIST OF INPUT WITH POSSIBLE ERRORMESSAGES  ************
 METASYMBOLS M1=# M2=/ M3=< M4=!

  BEGIN  END REPEAT WHILE DO UNTIL IF THEN ELSE
 KONST VAR  NAME  INTEGER ( )
 + * := , . ; !

   <PROGRAM< # <VARIABLE PART<   <STATEMENT SEQUENCE< END . !

  <VARIABLEPART< # VAR <DECLARATIONLIST< !

  <DECLARATIONLIST< # <DECLARATIONLIST< ;  <DECLARATION <

                    / <DECLARATION< !

 <DECLARATION< # INTEGER <NAMELIST< !

 <NAMELIST< # <NAMELIST< , NAME / NAME !

 <EXPRESSION< # <EXPRESSION< + <TERM < / <TERM< !

 <TERM< # <PRIMARY< * <TERM< / <PRIMARY< !

 <PRIMARY< # ( <EXPRESSION< ) / NAME / KONST  !

 <IFCLAUSE< # IF <EXPRESSION< THEN !

 <TRUEPART< # <IFCLAUSE< <CLOSED STATEMENT< ELSE !

  <WHILE< # WHILE !

  <WHILECLAUSE< # <WHILE<  <EXPRESSION< DO !

 <REPEAT< # REPEAT !

 <STATEMENT SEQUENCE<# BEGIN /  <STATEMENT SEQUENCE<

                     < STATEMENT< ; !

 <CLOSED STATEMENT< # <REFERENCE< :=<EXPRESSION<

          / <STATEMENT SEQUENCE< END !

 <REFERENCE< # NAME !

  <STATEMENT*< # <CLOSED STATEMENT<

           / <TRUEPART< <STATEMENT*<

          / <IF CLAUSE< <CLOSED STATEMENT<

          / <WHILE CLAUSE< <STATEMENT*<

          / <REPEAT< <STATEMENT< UNTIL <EXPRESSION< !

<STATEMENT< # <STATEMENT*< ! !

*******************************  END OF LIST ********************************
```

********** GRAMMARCHECKS **********

IT HAS BEEN CHECKED THAT ALL NONTERMINALS
EXCEPT THE GOALSYMBOL APPEAR IN BOTH
LEFT AND RIGHTSIDE OF A PRODUCTION

IT HAS BEEN CHECKED THAT THERE
EXIST NO IDENTICAL PRODUCTIONS

<STATEMENT> HAS BEEN SUBSTITUTED BY <STATEMENT*>
THE GRAMMER HAS BEEN MODIFIED FOR SIMPLE CHAINS
USING THE ABOVE STANDING SUBSTITUTIONS

IT HAS BEEN CHECKED THAT ALL NONTERMINALS CAN
PRODUCE A STRING OF ONLY TERMINAL SYMBOLS

IT HAS BEEN CHECKED THAT NO NONTERMINAL
CAN PRODUCE THE EMPTY STRING

IT HAS BEEN CHECKED THAT NO NONTERMINAL
IS BOTH LEFT AND RIGHT RECURSIVE

********** THE GRAMMAR BEFORE CONSTRUCTION OF THE LR0 **********

```
 1   <PROGRAM> ::= <VARIABLEPART> <STATEMENTSEQUENCE> END .

 2   <VARIABLEPART> ::= VAR <DECLARATIONLIST>

 3   <STATEMENTSEQUENCE> ::= BEGIN
 4                         / <STATEMENTSEQUENCE> <STATEMENT*> ;

 5   <DECLARATIONLIST> ::= <DECLARATIONLIST> ; <DECLARATION>
 6                       / <DECLARATION>

 7   <DECLARATION> ::= INTEGER <NAMELIST>

 8   <NAMELIST> ::= <NAMELIST> , NAME
 9                / NAME

10   <EXPRESSION> ::= <EXPRESSION> + <TERM>
11                  / <TERM>

12   <TERM> ::= <PRIMARY> * <TERM>
13            / <PRIMARY>

14   <PRIMARY> ::= ( <EXPRESSION> )
15               / NAME
16               / KONST

17   <IFCLAUSE> ::= IF <EXPRESSION> THEN

18   <TRUEPART> ::= <IFCLAUSE> <CLOSEDSTATEMENT> ELSE

19   <CLOSEDSTATEMENT> ::= <REFERENCE> := <EXPRESSION>
20                       / <STATEMENTSEQUENCE> END

21   <WHILE> ::= WHILE

22   <WHILECLAUSE> ::= <WHILE> <EXPRESSION> DO

23   <REPEAT> ::= REPEAT

24   <REFERENCE> ::= NAME

25   <STATEMENT*> ::= <CLOSEDSTATEMENT>
26                  / <TRUEPART> <STATEMENT*>
27                  / <IFCLAUSE> <CLOSEDSTATEMENT>
28                  / <WHILECLAUSE> <STATEMENT*>
29                  / <REPEAT> <STATEMENT*> UNTIL <EXPRESSION>
```

****

****

THE GRAMMAR IS SLR1

****

********** COMPILER ERROR MESSAGES **********

ERRORNO :					
0	** SPECIAL ERROR **				
ERRORNO :	EXPECTED SYMBOL:				
1 :	VAR				
2 :	INTEGER				
3 :	NAME				
4 :	;	BEGIN	,		
5 :	BEGIN	;			
6 :	BEGIN				
7 :	END WHILE	IF	REPEAT	NAME	BEGIN
8 :	(	NAME	KONST		
9 :	* ;	+ UNTIL	) ELSE	THEN	DO
10 :	)	+			
11 :	DO	+			
12 :	THEN	+			
13 :	:=				
14 :	ELSE	;	UNTIL	+	
15 :	IF	REPEAT	NAME	BEGIN	WHILE
16 :	;	UNTIL	ELSE		
17 :	UNTIL				
18 :	;	UNTIL	+		
19 :	;				
20 :	.				
21 :	BOBS-END				

****

Eksempel på indsættelse af semantik

I den producerede parser fra det foregående eksempel er der indsat semantiske procedurer. Disse procedurer genererer BPL-assembly-kode, efter følgende regler: Celle 0 indeholder konstanten 0, celle 1 indeholder konstanten 1. Fra celle 2 og fremad allokeres plads til de variable, således at den først erklærede variabel placeres i celle 2, den næste i celle 3 osv. Herefter starter den egentlige kode, og til sidst allokeres plads til konstanter. For nærmere tydning af koden henvises til relevant litteratur om BPL (f.eks. [10]).

De næste sider er en pseudo-listning af den producerede parser med indsat semantik. Kun de statements, der er indsat er listet. Resten af parseren er symboliseret med prikker.

```
CONST
    .
    .
    .
    .
    .
MEMBUFL=100;
NOOFCST=40;
STKMAX=30;

TYPE
    .
    .
    .
    .
ADRESS=0..4095;
ACTIONTYPE=(Z,AC,V,E,IC);
OPTYPE=(AS,VS,NOOP,NEQUAL,ADD,MULT,STORE,DELETE,JP,JPZ,RETURN);

VAR
    .
    .
    .
    .
    .
CODESTART,LASTCST,
ADRES,M,C,STKTOP :INTEGER;
CHH: CHAR;
MEM: ARRAY[0..MEMBUFL] OF PACKED
     RECORD   ACTION: ACTIONTYPE;
        OPERATION: OPTYPE;
        ADRESSE: ADRESS;
     END;

ADRSTACK: ARRAY[1..STKMAX] OF ADRESS;

TABLE: ARRAY[0..NAMENOMAX] OF PACKED
        RECORD
             TYPES: (DEFINED,UNDEFINED);
             VADRES: ADRESS;
        END;

CSTTABLE: ARRAY[1..NOOFCST] OF PACKED
          RECORD CONSTANT: -77777B..77777B;
               FIRST: 0..MEMBUFL;
           END;

VALUE # VALUE STATEMENT TIL INITIALISERING AF TABELLER
       (FRA GENERATOREN) !


    .
    .
    .
    .
    .
    .

FUNCTION AFSTAK(I:INTEGER #NOT USED!):INTEGER;
```

```
BEGIN
   AFSTAK:=ADRSTACK[STKTOP];
    STKTOP:=STKTOP-1;
END;

PROCEDURE SETFORWARDADR(ADR: ADRESS);
BEGIN
   WITH MEM[AFSTAK(0)]  DO
   BEGIN
        ACTION:=AC;
      OPERATION:=NOOP;
      ADRESSE:=ADR;
   END;
END # SETFORWARDADR ! ;

PROCEDURE REMEMBERADRES;
BEGIN
   STKTOP:=STKTOP+1;
   ADRSTACK[STKTOP]:=ADRES+1;
END;


FUNCTION CSTADRES(C :INTEGER):INTEGER;
# CSTADRES GENERERER ADRESSEN HVOR KONSTANTEN C ER LAGRET
 I DEN PRODUCEREDE KODE !
BEGIN
   IF (C=0)@(C=1) THEN
   BEGIN
      CSTADRES:=C; GOTO 20;
   END;

   FOR I:=1 TO LASTCST DO
   IF CSTTABLE[I].CONSTANT=C THEN GOTO 10;
   LASTCST:=LASTCST+1;
   WITH CSTTABLE[LASTCST] DO
   BEGIN
      CONSTANT:=C;
      FIRST:=ADRES+1;
      CSTADRES:=0;
      GOTO 20;
   END;
 10:
   WITH CSTTABLE[I] DO
   BEGIN
      CSTADRES:=FIRST;
      FIRST:=ADRES+1;
   END;
  20:
END;

PROCEDURE GENERATE(OP:OPTYPE; A:ADRESS);
BEGIN
   ADRES:=ADRES+1;
   IF ADRES>MEMBUFL THEN MARKERROR(30);
   WITH MEM[ADRES] DO
   BEGIN
         CASE OP OF
          AS: BEGIN ACTION:=AC;
               OPERATION:=NOOP;
              END;
```

```
          VS: BEGIN  ACTION:=V;
                     OPERATION:=NOOP;
                 END;
       NEQUAL,ADD,MULT,STORE,DELETE,JP,JPZ,RETURN:
           BEGIN ACTION:=E;
                 OPERATION:=OP;
           END;
         END;
      ADRESSE:=A;
   END;
END;


PROCEDURE CODE(PRODNR:INTEGER);
VAR
   I:INTEGER;
BEGIN
   CASE PRODNR OF
17,22: # <IFCLAUSE> ::= IF <EXPRESSION> THEN
       <WHILECLAUSE> ::= <WHILE> <EXPRESSION> DO !
      BEGIN
         REMEMBERADRES;
         ADRES:=ADRES+1;
         GENERATE(JPZ,0);
      END;

18:   # <TRUEPART> ::= <IFCLAUSE> <CLOSEDSTATEMENT> ELSE !
      BEGIN
         SETFORWARDADR(ADRES+3);
         REMEMBERADRES;
         ADRES:=ADRES+1;
         GENERATE(JP,0);
      END;

21,23:# <WHILE> ::= WHILE
        <REPEAT> ::= REPEAT !
      REMEMBERADR;

19:   # <CLOSEDSTATEMENT> ::= <REFERENCE> := <EXPRESSION> !
      GENERATE(STORE,0);

26,27: # <STATEMENT*> ::= <TRUEPART> <STATEMENT*>
                  / <IFCLAUSE> <CLOSEDSTATEMENT> !
      SETFORWARDADR(ADRES+1);

28:    # <STATEMENT*> ::= <WHILECLAUSE> <STATEMENT*> !
      BEGIN
         SETFORWARDADR(ADRES+3);
         GENERATE(AS,AFSTAK(0));
         GENERATE(JP,0);
     END;

8,9:   # <NAMELIST> ::= NAMELIST> , NAME  / NAME !
      BEGIN
         IF NAME > NAMENOMAX THEN MARKERROR(40) ELSE
         WITH TABLE(NAME) DO
         IF TYPES'UNDEFINED THEN MARKERROR(41) ELSE
         BEGIN
         TYPES:=DEFINED; ADRES:=ADRES+1;
         VADRES:=ADRES;
```

```
        END;
      END;

15:   # <PRIMARY> ::= NAME !
      WITH TABLE(NAME) DO
      IF TYPES=UNDEFINED THEN MARKERROR(42)
      ELSE GENERATE(VS,VADRES);

10:    # <EXPRESSION> ::= <EXPRESSION> + <TERM> !
      GENERATE(ADD,0);

12:   # <TERM> ::= <PRIMARY> * <TERM> !
      GENERATE(MULT,0);

2:    # <VARIABLEPART> ::= VAR <DECLARATIONLIST> !
      CODESTART:=ADRES+1;

16:   #<PRIMARY> ::= KONST !
      BEGIN
         IF KONST(KONSTNO)="B" THEN
        BEGIN  M:=8;
          KONSTNO:=KONSTNO-1
        END ELSE M:=10;
        C:=0;
        FOR I:=1 TO KONSTNO DO
        BEGIN
           CHH:=KONST(I);
           IF (CHH?"0")&(CHH\"9") THEN
           C:=C*M+INT(CHH)-27
           ELSE MARKERROR(43);
        END;
         GENERATE(VS,CSTADRES(C));
      END;

24:   # <REFERENCE> := NAME !
      WITH TABLE(NAME) DO
      IF TYPES=UNDEFINED THEN MARKERROR(42)
      ELSE GENERATE(AS,VADRES);

29:   #<STATEMENT*> ::=<REPEAT> <STATEMENT*> UNTIL <EXPRESSION> !
      BEGIN
         GENERATE(AS,AFSTAK(0));
         GENERATE(JPZ,0);
      END;

1:    BEGIN
         GENERATE(RETURN,0);
         # ALLOKERING AF KONSTANTER OG UDSKRIVNING AF KODEN !
         FOR I:=1 TO LASTCST DO
         WITH CSTTABLE(I) DO
         BEGIN
            ADRES:=ADRES+1;
            WITH MEM(ADRES) DO
            BEGIN
               ACTION:=IC;
               ADRESSE:=CONSTANT;
            END;
            REPEAT
               WITH MEM(FIRST) DO
               BEGIN
```

```
                    FIRST:=ADRESSE;
                    ADRESSE:=ADRES;
                END;
            UNTIL FIRST=0;
        END;
        # SLUT PAA ALLOKERING AF KONSTANTER !

        WRITE(EOL,EOL,EOL);
        FOR I:=CODESTART TO ADRES DO
        WITH MEM(I) DO
        BEGIN
            WRITE(EOL," ");
            CASE ACTION OF
            Z: ;
            AC: WRITE("  A     .",ADRESSE," "," ");
            V: WRITE("  V     .",ADRESSE," "," ");
            E: BEGIN
                WRITE("  E     .");
                CASE OPERATION OF
                    ADD:    TEXT("ADD,VS,VS    ");
                    MULT:   TEXT("MULT,VS,VS   ");
                    STORE:  TEXT("STORE,VS,AS  ");
                    NEQUAL:TEXT("NEQUAL,VS,VS");
                    JP:     TEXT("JP,AS        ");
                    JPZ:    TEXT("JPZ,AS       ");
                    RETURN:TEXT("RETURN        ");
                END;
               END;
              IC: WRITE("  I",ADRESSE," "," ");
            END;
            WRITE(";",I,EOL);
        END;
    END;

0,3,4,5,6,7,11,13,14,20,25: ; #NOOP!
    END;
END # END CODE ! ;

BEGIN  # MAIN PROGRAM !
    STKTOP:=0;
    ADRES:=1;
    LASTCST:=0;
    FOR I:=0 TO NAMENOMAX DO TABLE(I).TYPES:=UNDEFINED;

        .
        .
        .
        .
        .
        .
END.
```

Eksempel på output fra den nye parser.

```
VAR
   INTEGER I,J,FIRST,LAST;
   INTEGER A,B
BEGIN
   IF I+J THEN I:=I+1;
   WHILE FIRST DO
   BEGIN J:=J+1;
      REPEAT A:=A+B*(I+LAST)
      UNTIL J;
      IF J THEN I:=I+1 ELSE I:=I+2;
   END;
END.
```

```
   V    .          2  ;          9
   V    .          3  ;         10
   E    .  ADD,VS,VS  ;         11
   A    .         19  ;         12
   E    .  JPZ,AS     ;         13
   A    .          2  ;         14
   V    .          2  ;         15
   V    .          1  ;         16
   E    .  ADD,VS,VS  ;         17
   E    .  STORE,VS,AS;         18
   V    .          4  ;         19
   A    .         56  ;         20
   E    .  JPZ,AS     ;         21
   A    .          3  ;         22
   V    .          3  ;         23
   V    .          1  ;         24
   E    .  ADD,VS,VS  ;         25
   E    .  STORE,VS,AS;         26
   A    .          6  ;         27
   V    .          6  ;         28
```

```
V    .          7   ;        29
V    .          2   ;        30
V    .          5   ;        31
E    .  ADD,VS,VS   ;        32
E    .  MULT,VS,VS  ;        33
E    .  ADD,VS,VS   ;        34
E    .  STORE,VS,AS ;        35
V    .          3   ;        36
A    .         27   ;        37
E    .  JPZ,AS      ;        38
V    .          3   ;        39
A    .         49   ;        40
E    .  JPZ,AS      ;        41
A    .          2   ;        42
V    .          2   ;        43
V    .          1   ;        44
E    .  ADD,VS,VS   ;        45
E    .  STORE,VS,AS ;        46
A    .         54   ;        47
E    .  JP,AS       ;        48
A    .          2   ;        49
V    .          2   ;        50
V    .         57   ;        51
E    .  ADD,VS,VS   ;        52
E    .  STORE,VS,AS ;        53
A    .         19   ;        54
E    .  JP,AS       ;        55
E    .  RETURN      ;        56
I               2   ;        57
```

APPENDIX D

Litteraturliste.

[1] DeRemer, F.L.
"Practical Translation for LR(k) Languages"
Ph.D. Thesis, Massachusetts Institute of Technology,
Cambridge, Mass. August 1969.

[2] DeRemer, F.L.
"Simple LR(k) Grammars"
CACM, p. 453-459. (14,7,1971)

[3] Knuth, D.E.
"On the Translation of Languages from left to right"
Inf. and Cont. p. 607-639. (Oct. 1965)

[4] McKeeman, W.M. et al.
"A Compiler Generator"
(Prentice/Hall) 1970.

[5] Wirth and Weber
"Euler: A Generalization of Algol and its Formal Definition I, II"
CACM p. 13-25, 88-99. (9,1-2, 1966).

[6] Gries, D.J.
"Compiler Construction for Digital Computers"
Wiley. Toronto, Ontario, 1969.

[7] Lalonde, W.R.
"An Efficient LALR-Parser-Generator"
Tech. Report CSRG-2, University of Toronto
Toronto, Ontario, 1971.

[8] Wirth, N.
"A Basic Course on Compiler Principles"
BIT 9 (1969) p. 362-386.

[9] Horning, J.J., Lalonde, W.R.
"Empirical Comparison of LR(k) and Precedence Parsers"
Tech. Report CRSG-1. University of Toronto
Toronto, Ontario, 1970.

[10] Jensen, E.S. & J.E. Thomsen
"Implementation of BPL on RIKKE/KAROLINE"
RECAU-72-17.

[11] BOBS
Bent Bæk Jensen
Ole Lehrmann Madsen
Bent Brun Kristensen
Søren Henrik Eriksen
"Obligatorisk Datalogi 2 opgave, SLR(1)-Parsergenerator og Parser"
DAIMI, Aarhus Universitet.

/km