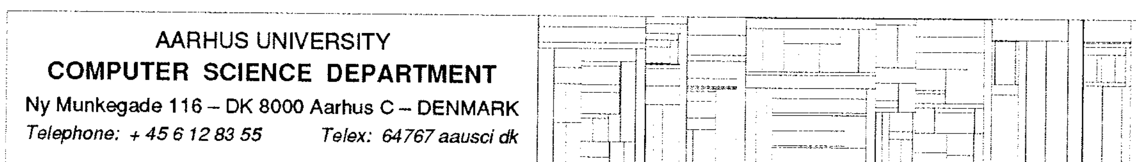
ISSN 0105-8517

THE BOBS-SYSTEM

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

DAIMI PB – 71
March 1977

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 6 12 83 55 *Telex: 64767 aausci dk*



THE BOBS-SYSTEM

by

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

The BOBS-System is an LALR(1) parser-generator. This paper is a user manual for the system: consisting of a general description of the system, a reference manual, and a summary of parsing terminology. These sections can be read without knowledge of parsing theory.

Furthermore, the implementation of the system is described. This is done by giving references to literature containing descriptions of some of the algorithms used, and by giving abstract algorithms for other parts of the system. This section requires that the reader is familiar with LR-parsing.

* A/S Regnecentralen, Aarhus
** KTAS, København
*** Department of Computer Science, Institute of Electronic Systems, Aalborg University Center, DK-9000 Aalborg

CONTENTS

INTRODUCTION 1

PART I 2

1. The history of the BOBS-system 2

2. Global design of the system 3

2.1 The parser-generator 3

2.1.1 Specification of the source grammar 3

2.1.2 Grammar checks and grammar transformations 3

2.1.3 Construction of the LALR(1)-tables 4

2.1.4 Output from the parser-generator 4

2.1.5 SLR(1) 5

2.2 The skeleton compiler 5

2.2.1 The semantic interface 5

2.2.2 The parser part 5

3. Evaluation 6

PART II 8

4. User Manual 8

4.1 Notation 8

4.2 Syntax of input to the parser-generator 8

4.2.1 Metasymbols 8

4.2.2 Terminals 9

4.2.3 Stringch 10

4.2.4 Grammar-rule 10

4.2.5 Goalsymbol 10

4.2.6 Endcharacter 11

4.2.7 Comment 11

4.2.8 Options 11

4.3 Constants 13

4.4 Error messages 14

4.4.1 System errors 14

4.4.2 Errors according to the input grammar 14

4.5 Examples on using the parser-generator 14

5. The Skeleton-compiler 22

5. 1 Input/output of the Skeleton-compiler 23

5. 2 Adding semantics 23

5. 2. 2 Using NAME, KONST, and STRING 24

5. 2. 3 Using the semantic stack 24

5. 3 Error recovery 27

5. 4 Example 29

PART III 39

6. Program description 39

6. 1 Description of the Parser-generator 39

6. 1. 1 The construction of the parse tables 40

6. 1. 2 The LALR(1) lookahead algorithm 41

6. 2 Description of the Skeleton-compiler 44

6. 2. 1 The parsing algorithm 44

6. 2. 2 The recovery algorithm 48

PART IV 52

7. Summary of terminology 52

7. 1 Context free grammar 52

7. 2 Parser and parser-generator 54

7. 3 LR(k) grammar 55

7. 4 Practical LR-grammars 65

REFERENCES 71

APPENDIX 74

A 1 The BOBS-System at RECAU 74

A 2 The BOBS-System at DAIMI 75

B A (minor) extension to the BOBS-system 76

PREFACE to the Third Edition

This third edition is identical to the first and second edition except that some minor errors have been corrected, appendix B has been added and section 6. 1. 2 has been replaced by a reference to a more appropriate source.

INTRODUCTION

This paper is a description of a parser-generator system called the BOBS-system.

We consider the LR-grammars [3]. These are, in ascending order of complexity, LR(0), SLR(k), LALR(k), L(m) R(k) and LR(k) as defined in [1, 2].

The BOBS-system is an implementation of a parser-generator in the programming language PASCAL [12] for the SLR(1) and LALR(1) grammars.

This paper is divided into four parts: part 1 is a general description of the system; part 2 is a user manual; in part 3 we describe the program organization and some algorithms of special interest; and part 4 contains an introduction to the terminology which we use. (Readers not familiar with the terminology should read this part first.)

PART I

1. The history of the BOBS-system

The work was started as an undergraduate project in 1971 under the guidance of Peter Kornerup. The aim of the project was to implement a parser-generator system for SLR(1)-grammars. This was fulfilled in March 1972. The first version was implemented in the earliest version of the programming language Pascal [6] on a CDC 6400.

A revised version was released in December 1972, and a user manual [7] appeared in March 1973. At the same time a short introductory description of the system [8] was published.

During 1973 the system was extended and modified [9] as a result of our experiences in using the system. A new lexical analyser and a new error-recovery method were implemented. A new look-ahead algorithm was constructed and implemented, extending the system to accept LALR(1)-grammars.

During 1974 a new user manual was written. In October 1974 this manual and system description [8] were published in [10].

Since December 1974 a modified version of the system has been distributed from the University of Texas at Austin by W.F. Burger. This version is called "BOBSW - A Parser Generator" [11].

In October 1976 a new modified version of the system was finished. First of all the parser generator has been translated (more or less) mechanically into standard Pascal [12], so that the system is no longer dependent on the earliest Pascal compiler on the CDC 6400. Next the parser generator has been extended to handle grammars, whose grammar rules contain empty productions, without eliminating such productions. A new skeleton compiler has been implemented. This means new lexical, syntax and error-recovery algorithms, and implementation of a user semantic stack.

2. Global design of the system

The system consists of two programs:

- the parser-generator,
- the skeleton compiler.

2.1 The parser-generator

The parser-generator is divided into the following modules:

- input of the source grammar,
- grammar checks and grammar transformations,
- generation of the LALR(1)-tables.

2.1.1 Specification of the source grammar

The source grammar must be specified in a slightly modified BNF (Backus Naur Form).

2.1.2 Grammar checks and grammar transformations

It has turned out that the implemented grammar checks and transformations are very useful when designing a grammar for a language.

The system cannot produce the LALR(1)-tables for an ambiguous grammar or grammar containing unused nonterminal symbols.

The system performs the following grammar checks:

Test for unused nonterminal symbols:

- the system checks that every nonterminal except the goal-symbol appears on both the left and the right side of a production,
- the system checks that all nonterminals can be derived from the goal symbol (the start symbol),
- the system checks that all nonterminals can derive a string only containing terminals.

Test for ambiguity:

- the system checks whether any nonterminal is both left and right recursive. If so, the grammar is ambiguous.

The system can perform the following transformations on the grammar:

- If identical productions exist, the grammar can be modified by removing the unnecessary productions.
- If any nonterminal can derive the empty string, the grammar can be modified by eliminating this production. The modified grammar generates the same language.
- If there exist single productions, the grammar can be modified by eliminating all such. In a single production the left and right side both consist of only a single nonterminal, and the left side nonterminal does not appear on the left side of any other production.

2.1.3 Construction of the LALR(1)-tables

The first step is to construct the LR(0) parse tables and check whether the grammar is LR(0) or not. If this fails, the parse tables are extended by means of LALR(1) lookahead. If this fails too, the system reports the parse tables in which the lookahead information is insufficient. The user must then change the grammar.

If the grammar happens to be LALR(1), the parse tables are compressed through a series of various optimizations.

2.1.4 Output from the parser-generator

The system delivers the following output:

- a listing of the source grammar, exactly as the user has written it. Any error according to the input syntax is marked;
- the results of the above mentioned grammar checks and transformations;
- the modified grammar written in BNF. Each production is assigned a unique number;
- a description of any parse tables which are not LALR(1);

- an error message table for use by the user when parsing an erroneous string;
- a file containing the parse tables and a file containing the skeleton compiler (a Pascal program).

In addition there exist several other output facilities, mentioned in Section 4.2.7 but they are of little interest here.

2.1.5 SLR(1)

As mentioned in Section 1 the system was originally designed for SLR(1) grammars. The possibility of using SLR(1)look-aheads instead of LALR(1) still exists.

2.2 The skeleton compiler

The second part of the system is a Pascal program, which is delivered as output from the parser-generator. The unmodified skeleton compiler will check the syntax of an input string written in the language defined by the source grammar. However, the user may add semantic actions to the skeleton compiler.

The skeleton compiler is divided into the following modules:

- the semantic interface
- the parser part
    - lexical analysis
    - syntax analysis
    - error recovery.

2.2.1 The semantic interface

When a reduction is performed, a procedure "CODE" is called with the production (reduction) as a parameter. The user may then decide what semantic actions to perform. This is done by writing the body of the procedure "CODE". The user may, of course, add new procedures and declarations.

2.2.2 The parser part

In this part of the program the user does not have to change anything.

2.2.2.1 Lexical analysis

Characters from the input are read and collected into single logical items called tokens. A token corresponds to a terminal symbol of the grammar.

2.2.2.2 Syntax analysis

The syntax analyser uses the parse tables constructed by the generator to parse the string of tokens delivered from the lexical analyser.

2.2.2.3 Error recovery

If an error is discovered during parsing, an error recovery algorithm is called. The purpose of the recovery algorithm is to mark the symbol causing the error detection, to recover from the erroneous situation, and to initialize a continuation of the parsing.

3. Evaluation

The system has been used in a variety of different projects. A compiler for the language Pascal [6, 13] has been based on the system. The Pascal grammar consists of more than 250 productions and the constructed LALR(1) tables occupy about 1000 60 bit words on a CDC 6400. Compared with the Zürich Pascal Compiler (version 6. Sept. 72) the core requirements and execution time are nearly the same.

Inside the department the system has been used for various compilers and assemblers. It is also used in a compiler course, in which the students have to write a small compiler.

At the Danish Data Archive (an institution under the Danish Social Science Research Council) and at the Institute of Economics, University of Aarhus, a modified version of the system is used to implement interactive special purpose languages, which are designed to ease the use of libraries of statistical programs, the handling of files and the controlling of data bases. This system consists of about 20 grammars each containing from 160 to 250 productions.

As the work has been moving along,new projects have arisen. Automatic error recovery in LR-parsing has been studied [14]. Also problems of de-

fining semantics have been studied. One project [15] was to extend the system with the Oxford semantics [16]. Furthermore the use of attribute grammars in practical translator writing systems has been studied [17].

We conclude that the system is usable in practice and that experience has shown that it is easy to modify grammars to become LALR(1), even for users who are not familiar with LR-parsing theory.

PART II

4. User Manual

4.1 Notation

The specification of the syntax of input to the parser generator is given in extended BNF. In this notation a set of additional metasymbols is introduced. These may be used in the specification in the following way:

{ } clauses enclosed in these parantheses are grouped into a single clause,

* the clause preceeding this symbol may be repeated zero or more times,

+ the clause preceeding this symbol may be repeated one or more times,

? the clause preceeding this symbol is optional.

Finally the metasymbol ::= is replaced by the symbol → .

4.2 Syntax of input to the parser generator

<PARSER - GENERATOR - INPUT> →
{ <OPTIONLIST> } ?
{ <METASYMBOL - DEFINITION> } ?
{ <TERMINAL - DEFINITION> }
{ <STRINGCH - DEFINITION> } ?
{ <GOALSYMBOL - DEFINITION> } ?
{ <COMMENT - DEFINITION> } ?
{ <GRAMMAR - RULE> | <METASYMBOL - DEFINITION> } *
<ENDCH>

4.2.1 Metasymbols

<METASYMBOL- DEFINITION> → METASYMBOLS <M1> <M2> <M3> <M4>
<M1> → M1 = <CH>
<M2> → M2 = <CH>
<M3> → M3 = <CH>
<M4> → M4 = <CH>

<CH> is any character other than a letter, a digit or a space.

The metasymbols must be different. The correspondence to the use in BNF is:

M1 works as ::=

M2 works as |

M3 works as < and >

M4 indicates the termination of a sequence of alternatives in a grammar rule.

Default metasymbols are:

M1==, M2=/, M3=<, and M4=;

In the following M1, M2, M3, and M4 denote the current metasymbols.

4.2.2 Terminals

<TERMINAL - DEFINITION> → <TERMINAL>* M4

All terminal symbols used in the grammar must be listed. A terminal symbol consists of at most 10 characters. The character set has been divided into two groups:

- letter and digits
- all other characters except space

All the characters forming a terminal must belong to the same set of the above groups. Terminals consisting of symbols from group 1 must start with a letter.
No terminal may contain the current M4.
The terminal symbols in the list must be delimited by spaces and/or end-of-lines.

The following terminals have a special interpretation. If they are used in the grammar, they must be listed among the other terminals.

EMPTY denotes the empty string.

NAME denotes an identifier.(A sequence of letters and digits with the first symbol being a letter.)

KONST denotes a constant. (A sequence of letters and digits with the first symbol being a digit.)

STRING denotes a string constant. (A string is a sequence of characters surrounded by a string-escape-character. If

the string-escape-character is used in the string, it must be written two times per occurrence.

ERROR denotes an error-symbol. The use of the error-symbol is explained in Section 5.3.

4.2.3 Stringch

<STRINGCH - DEFINITION> → STRINGCH = <CH> M4

Defines the string-escape-character to be the character <CH>. It must not be contained in any other terminal symbol. No default value exists.

4.2.4 Grammar-rule

<GRAMMAR - RULE> → <NONTERMINAL> M1 <ALTERNATIVE>
{ M2 <ALTERNATIVE> } * M4
<ALTERNATIVE> → { <NONTERMINAL> | <TERMINAL> }+
<NONTERMINAL> → M3 a sequence of characters M3

The sequence of characters must not contain the current M3. Spaces are skipped. A nonterminal may consist of up to 30 characters. Terminals and nonterminals in a rule must be separated by spaces or end of lines.

If EMPTY is used it must be the only symbol in that alternative.

The terminals in a grammar rule must not contain any of the metasymbols currently defined. If they have to, the metasymbols must be redefined. It is not necessary for all alternatives to the same lefthandside of a grammar rule to be defined at the same time, they may be defined later.

4.2.5 Goalsymbol

<GOALSYMBOL - DEFINITION> → GOALSYMBOL = <NONTERMINAL> M4

Defines the nonterminal to be the goalsymbol of the grammar. If this command is not present, the first nonterminal met among the grammar rules is assumed to be the goalsymbol.

The parser generator always adds the following grammar rule (production no. 0):

<BOBS - GOAL> → <GOALSYMBOL> END-OF-FILE

4.2.6 Endcharacter

<ENDCH> → M4

The input to the parser generator must be terminated by the currently defined M4.

4.2.7 Comment

<COMMENT> → COMMENT = <COMMENT - BEGIN> M4 <COMMENT - END> M4
<COMMENT - BEGIN> → <TERMINAL>
<COMMENT - END> → a sequence of characters

<TERMINAL> must be defined in the <TERMINAL - DEFINITION> (4.2.2). The sequence of characters of <COMMENT - END> must not include the current M4. Spaces are skipped.

In the input string to be parsed the following is considered to be a comment:

<COMMENT - BEGIN> any sequence of characters (except <COMMENT - END>)
<COMMENT - END>

See also Section 5.1.

4.2.8 Options

<OPTIONLIST> → OPTIONS (<OPTION - NUMBER>
{, <OPTION - NUMBER>} *)

<OPTION - NUMBER> is an integer. Most of the options are for test purposes only, and some may cause an error in the generated parser. The options are

implemented as switches, which means that if the same number is used twice, the option is returned to the original position.

For most users, the rest of this section is of no interest, and may be skipped.

The following numbers are valid:

1 The internal representations of the terminal symbols are printed.

2 The LR(0) tables are printed.

3 The terminal symbols may be collected in sets where all terminals in such a set are given the same internal value. Let T1, T2, ..., TN be terminals. If in the list of terminals (4.2.2) you write:

T1 T2 M1 T3 M1 ... M1 TN M1

then T1, T2, ..., TN are all given the same internal value. There must be exactly one space between a terminal and M1. In all outputs, T1, T2, ..., TN-1 will appear as TN.

4 The length of the terminal symbols may be greater than 10 characters. All characters are significant but only the 10 first appear in output.

5 The terminal symbols may consist of characters from both character groups (see chapter 4.2.2). The terminal symbols must be separated by blanks or an end-of-line. (The lexical analyser in the skeleton compiler must be modified for these terminals.)

6 The internal values of the nonterminal symbols are printed.

7 The SLR(1) lookahead symbols for each nonterminal are printed. (Caution: option 27 must be used.)

12 The inadequate LR(0) tables are printed.

14 The internal form of the LR(0) tables is printed.

15 The internal form of the LR(0) tables and the generated lookback tables are printed.

16 The internal array "PROD" is printed.

17 The internal array "TILSTAND" is printed.

18 The terminal heads and tails which a nonterminal can produce are printed.

19 Test information of "VHRECURS" is printed.

20 The bit matrix of the grammar is printed.

21 Test information of "LOOKBACK" is printed.

22 The largest lookahead set is not removed from a lookahead table. (The table is treated like a lookahead-error table.)

23 The internal form of the productions is printed before the grammar is modified.

24 The internal form of the productions is printed after the grammar is modified.

25 No attempt is made to test for left and right recursion.

27 The grammar is treated as an SLR(1) grammar instead of an LALR(1) grammar. In this case one has to use option 31 too.

28 No attempt is made to remove the single productions.

29 Tables which have the same tail are not folded together.

30 The LR(0) items are printed in a readable form.

31 The empty string is eliminated from the grammar.

32 Files PARSIN and PARSOUT (see appendix A) are ignored. Only parse tables are produced (on file TABLES).

33, 34 See appendix B.

4.3 Constants

The following constants define the maximum sizes of the data structures in the parser-generator.

Most of the constants are totally internal to the program and should only be changed with care.

CONST1	Maximum number of productions.
CONST2	Maximum number of terminal and nonterminal symbols,
CONST3	Maximum size of array FCQ.
CONST4	Maximum size of array RHS.
CONST5	Maximum number of elements in a basis set of an LR(0) table.
CONST7	Maximum size of final parse tables.
CONST8	Maximum number of lookahead elements for a nonterminal.
CONST9	Maximum size of array STACK in procedure LR0.
CONST10	Maximum number of parse tables.
CONST11	Maximum number of nonterminals.
CONST12	Maximum number of terminals.
CONST13	Equal CONST2 DIV (SETMAX+1) + 1 [‡)]
CONST14	Maximum size of a parse table.
CONST16	Maximum size of array LL in procedure LALR LOOKAHEAD.
CONST18	Maximum number of options.

4.4 Error messages

Two types of error messages can occur:

- system errors
- errors according to the grammar.

4.4.1 System errors

One of the data structures in the parser generator has caused an error. The appropriate constant must be changed in the parser generator.

4.4.2 Errors according to the input grammar

There might be errors:

- in input to the parser generator,
- according to the grammar checks and grammar transformations,
- in the parse tables, which are not LALR(1).

4.5 Examples on using the Parser Generator

Following are two examples. First a complete example of input of a grammar and the produced output. The second example shows a grammar which is not LALR(1). See also the appendix.

---

‡) CONST SETMAX = 58 in the CDC version.

Example 4.1.

Input to the parser generator:

```
METASYMBOLS M1=# M2=$ M3=" M4=?
DECLARE IF THEN ELSE FI WHILE DO OD
READ WRITE ( ) EOL [ ]  =  <> + - / * (*'
. ; , : := EMPTY KONST NAME ERROR STRING ?
STRINGCH= '?
GOALSYMBOL=PROGRAM ?
COMMENT= (* ? *) ?
"PROGRAM" # "DECLARATION" "STATEMENT-SEQ". ?
"DECLARATION" # DECLARE "VARLIST" ; ?
"VARLIST" # "VARLIST" , "ITEM"  $ "ITEM"  ?
"ITEM" # NAME  $ NAME [ "CONSTANT" : "CONSTANT" ] ?
"STATEMENT-SEQ" # "STATEMENT-SEQ" ; "STATEMENT"  $  "STATEMENT"  ?
"STATEMENT" # EMPTY
    $ "VARIABLE"  := "EXP"
    $ IF "EXP" THEN "STATEMENT-SEQ" ELSE "STATEMENT-SEQ" FI
    $ WHILE "EXP" DO "STATEMENT-SEQ" OD
    $ IF "EXP" THEN "STATEMENT-SEQ" FI
    $ READ( "VARIABLE" )  $  WRITE( "OUTPUT" ) $ EOL ?
"OUTPUT" # "EXP" $ STRING ?
"EXP" # "AEXP" "RELOP" "AEXP"  $  "AEXP" ?
"AEXP" # "AEXP" "ADDOP" "TERM" $ "TERM" ?
"TERM" # "TERM" "MULTOP" "PRIMARY" $ "PRIMARY" ?
"PRIMARY" # "VARIABLE" $ "CONSTANT" $ ( "EXP" ) ?
"VARIABLE" # NAME  $ NAME [ "EXP" ] ?
"RELOP" # = $ <> ?
"ADDOP" # + $ - ?
"MULTOP" # * $ / ?
"CONSTANT" # KONST $ + KONST $ - KONST ?
"PROGRAM" # ERROR ?
"DECLARATION" # DECLARE ERROR ; ?
"STATEMENT" # ERROR ?
"EXP" # ERROR ?
?
```

Output of the parser generator:

```
************ A LIST OF INPUT WITH POSSIBLE ERRORMESSAGES ************

      METASYMBOLS M1=# M2=$ M3=" M4=?
     DECLARE IF THEN ELSE FI WHILE DO OD
     READ WRITE ( ) EOL [ ]  =  <> + - / * (*'
     . ; , : := EMPTY KONST NAME ERROR STRING ?
     STRINGCH= '?
     GOALSYMBOL=PROGRAM ?
     COMMENT= (* ? *) ?
     "PROGRAM" # "DECLARATION" "STATEMENT-SEQ". ?
     "DECLARATION" # DECLARE "VARLIST" ; ?
     "VARLIST" # "VARLIST" , "ITEM"  $ "ITEM"  ?
     "ITEM" # NAME  $ NAME [ "CONSTANT" : "CONSTANT" ] ?
     "STATEMENT-SEQ" # "STATEMENT-SEQ" ; "STATEMENT"  $  "STATEMENT"  ?
     "STATEMENT" # EMPTY
         $ "VARIABLE"  := "EXP"
         $ IF "EXP" THEN "STATEMENT-SEQ" ELSE "STATEMENT-SEQ" FI
         $ WHILE "EXP" DO "STATEMENT-SEQ" OD
         $ IF "EXP" THEN "STATEMENT-SEQ" FI
         $ READ( "VARIABLE" )  $  WRITE( "OUTPUT" ) $ EOL ?
     "OUTPUT" # "EXP" $ STRING ?
     "EXP" # "AEXP" "RELOP" "AEXP"  $  "AEXP" ?
     "AEXP" # "AEXP" "ADDOP" "TERM" $ "TERM" ?
     "TERM" # "TERM" "MULTOP" "PRIMARY" $ "PRIMARY" ?
     "PRIMARY" # "VARIABLE" $ "CONSTANT" $ ( "EXP" ) ?
     "VARIABLE" # NAME  $ NAME [ "EXP" ] ?
     "RELOP" # = $ <> ?
     "ADDOP" # + $ - ?
     "MULTOP" # * $ / ?
     "CONSTANT" # KONST $ + KONST $ - KONST ?
     "PROGRAM" # ERROR ?
     "DECLARATION" # DECLARE ERROR ; ?
     "STATEMENT" # ERROR ?
     "EXP" # ERROR ?
     ?


****************************** 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
 EXISTS NO IDENTICAL PRODUCTIONS


 THE GRAMMER HAS BEEN CHECKED FOR
 SIMPLE CHAINS



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


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

```
******************************** THE GRAMMAR ********************************


  1    <PROGRAM> ::= <DECLARATION> <STATEMENT-SEQ> .
  2                / ERROR

  3    <DECLARATION> ::= DECLARE <VARLIST> ;
  4                    / DECLARE ERROR ;

  5    <STATEMENT-SEQ> ::= <STATEMENT-SEQ> ; <STATEMENT>
  6                      / <STATEMENT>

  7    <VARLIST> ::= <VARLIST> , <ITEM>
  8                / <ITEM>

  9    <ITEM> ::= NAME
 10             / NAME [ <CONSTANT> : <CONSTANT> ]

 11    <CONSTANT> ::= KONST
 12                 / + KONST
 13                 / - KONST

 14    <STATEMENT> ::= EMPTY
 15                  / <VARIABLE> := <EXP>
 16                  / IF <EXP> THEN <STATEMENT-SEQ> ELSE <STATEMENT-SEQ> FI
 17                  / WHILE <EXP> DO <STATEMENT-SEQ> OD
 18                  / IF <EXP> THEN <STATEMENT-SEQ> FI
 19                  / READ ( <VARIABLE> )
 20                  / WRITE ( <OUTPUT> )
 21                  / EOL
 22                  / ERROR

 23    <VARIABLE> ::= NAME
 24                 / NAME [ <EXP> ]

 25    <EXP> ::= <AEXP> <RELOP> <AEXP>
 26            / <AEXP>
 27            / ERROR

 28    <OUTPUT> ::= <EXP>
 29               / STRING

 30    <AEXP> ::= <AEXP> <ADDOP> <TERM>
 31             / <TERM>

 32    <RELOP> ::= =
 33              / <>

 34    <ADDOP> ::= +
 35              / -

 36    <TERM> ::= <TERM> <MULTOP> <PRIMARY>
 37             / <PRIMARY>

 38    <MULTOP> ::= *
 39               / /

 40    <PRIMARY> ::= <VARIABLE>
 41                / <CONSTANT>
 42                / ( <EXP> )

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

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

      THE GRAMMAR IS LALR1

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


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


   ERRORNO : 0         ** SPECIAL ERROR **

   ERRORNO :           EXPECTED SYMBOL:

         1 :           ERROR       DECLARE

         2 :           ERROR       NAME

         3 :           KONST       +           -

         4 :           KONST

         5 :           :

         6 :           ]

         7 :           ;

         8 :           ;           ,

         9 :           NAME

        10 :           ERROR       (           NAME        KONST       +
                       -

        11 :           (           NAME        KONST       +           -

        12 :           )

        13 :           (

        14 :           STRING      ERROR       (           NAME        KONST
                       +           -

        15 :           DO

        16 :           THEN

        17 :           :=

        18 :           ELSE        FI          ;

        19 :           FI          ;

        20 :           OD          ;

        21 :           .           ;

        22 :           -EOF-

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

Example 4.2

The symbol FI has been removed from productions 16 and 18. This makes the grammar non LALR(1). Only the grammar and the non LALR tables are shown.

There are two tables (or states) which are not LALR(1). In the first table the symbols ELSE and ; cause the errors. For instance if the next symbol on input is ELSE, the parser cannot decide whether it should reduce by production 18 or continue reading according to production 16.

In the second table it is ; which gives problems.

THE GRAMMAR

```
 1   <PROGRAM> ::= <DECLARATION> <STATEMENT-SEQ> .
 2               / ERROR

 3   <DECLARATION> ::= DECLARE <VARLIST> ;
 4                   / DECLARE ERROR ;

 5   <STATEMENT-SEQ> ::= <STATEMENT-SEQ> ; <STATEMENT>
 6                     / <STATEMENT>

 7   <VARLIST> ::= <VARLIST> , <ITEM>
 8               / <ITEM>

 9   <ITEM> ::= NAME
10            / NAME [ <CONSTANT> : <CONSTANT> ]

11   <CONSTANT> ::= KONST
12                / + KONST
13                / - KONST

14   <STATEMENT> ::= EMPTY
15                 / <VARIABLE> := <EXP>
16                 / IF <EXP> THEN <STATEMENT-SEQ> ELSE <STATEMENT-SEQ>
17                 / WHILE <EXP> DO <STATEMENT-SEQ> OD
18                 / IF <EXP> THEN <STATEMENT-SEQ>
19                 / READ ( <VARIABLE> )
20                 / WRITE ( <OUTPUT> )
21                 / EOL
22                 / ERROR

23   <VARIABLE> ::= NAME
24                / NAME [ <EXP> ]

25   <EXP> ::= <AEXP> <RELOP> <AEXP>
26           / <AEXP>
27           / ERROR

28   <OUTPUT> ::= <EXP>
29              / STRING

30   <AEXP> ::= <AEXP> <ADDOP> <TERM>
31            / <TERM>

32   <RELOP> ::= =
33             / <>

34   <ADDOP> ::= +
35             / -

36   <TERM> ::= <TERM> <MULTOP> <PRIMARY>
37            / <PRIMARY>

38   <MULTOP> ::= *
39              / /

40   <PRIMARY> ::= <VARIABLE>
41               / <CONSTANT>
42               / ( <EXP> )
```

```
*********************  A LIST OF NON-LALR1 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
                       ELSE               16           5
                       ;                   5           2

          REDUCTION   NO   18 IS PERFORMED IF THE NEXT INPUTSYMBOL
                      IS ONE OF THE FOLLOWING :
                      ELSE
                      OD
                      .
                      ;

          ----------------------------------------------------------------
          READING     CONTINUES IF THE NEXT INPUTSYMBOL
                      IS ONE OF THE FOLLOWING :
                      SYMBOL        PRODUCTION     SYMBOL NO
                       ;                   5           2

          REDUCTION   NO   16 IS PERFORMED IF THE NEXT INPUTSYMBOL
                      IS ONE OF THE FOLLOWING :
                      ELSE
                      OD
                      .
                      ;

**********************  END OF NON-LALR1 STATES  ********************


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

        THE GRAMMAR IS NOT LALR1
        THE PROGRAM STOPS

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

5. The Skeleton Compiler

When using the parser-generator, the Skeleton-compiler must reside on file PARSIN. The parser-generator then delivers the Skeleton-compiler with initialized constants on the file PARSOUT. The parse tables associated with the user's grammar are delivered on the file TABLES. It would have been more handy to incorporate the parse tables in the Skeleton-compiler as a set of initializations, but this is however not possible in Standard PASCAL.

The Skeleton-compiler consists of

- PROCEDURE PARSER,
- PROCEDURE CODE, and
- some global declarations.

The procedure PARSER is the major part of the Skeleton-compiler. It consists of procedures for doing:

- Lexical analysis,
- Context-free syntax analysis (parsing), and
- Error recovery.

We shall not discuss these procedures here, but the reader is referred to section 5.1 for a specification of how the input string to the lexical analyser must look. The use of the special tokens NAME, KONST, and STRING is explained in section 5.2.2. Error recovery is treated in section 5.3.

The procedure CODE is an almost empty procedure which has to be written by the user. CODE is called from the parser each time a reduction is performed during the parsing of a string. In this way CODE will act as an interface between the parser and the semantic part of a compiler based on the Skeleton-compiler.

Among the global declarations is a stack which may be used by the user (see section 5.2.3).

5.1 Input/Output of the Skeleton-compiler

Input to the Skeleton-compiler:

- On file INPUT: a string in the language generated by the grammar. Terminals in this string must be separated by spaces and/or end-of-lines. However two terminals may be concatenated if they are not in the same group of characters (see section 4.2.2). Terminals from group 2 may be concatenated if the concatenation does not together form the head of another terminal. Spaces and/or end-of-lines are only allowed as separators between terminals and are considered as blind characters. A comment may appear between any two terminals. It may be necessary to surround <COMMENT-BEGIN> by spaces in order to avoid that <COMMENT-BEGIN> concatenated with the preceeding terminal and/or the beginning of a comment forms the head of some terminal.

- On file TABLES: the parse tables of the grammar.

Output from the Skeleton-compiler (on file OUTPUT):

- A listing of the input string. Possible syntax errors in the string are marked (see section 5.3).

- A snapshot of the parse. Contains a general print-out of the steps in the parse of the actual input string. It is intended as an aid to the user and may be removed.

5.2 Adding Semantics

As mentioned, the procedure CODE is called from the parser each time a reduction is performed. CODE has as parameter the number of the applied production. The productions are numbered according to the listing produced by the parser-generator (see section 2.1.4).

Inside CODE, an appropriate action must be taken for each production in the grammar.

Besides filling out the body of CODE, the user is free to add new global declarations (procedures, variables, etc.).

As the parsing method is LR, the reductions will be performed in the order of a so-called right-parse. It is fundamental for using the BOBS-system, that the notion of a right-parse is understood. A definition of a right-parse is given in part IV. The snapshot of the parse may be an aid in understanding the order of the reductions.

5.2.2 Using NAME, KONST, and STRING

If the symbols NAME, KONST, and STRING are used on the right side of a production, then the user can get the string of characters actually comprising the NAME, KONST, or STRING. This is done by means of:

```
PROCEDURE GETSTRING (NO: INTEGER;
        VAR STR: STRING; VAR LENGTH: INTEGER);
```

where STRING = PACKED ARRAY[1..STRINGMAX] OF CHAR;

Suppose CODE is called with the number of the production

$$X_0 \rightarrow X_1 \ldots X_i \ldots X_n$$

If X_i is NAME, KONST, or STRING, then

GETSTRING(i,S,L) or GETSTRING(-(n-i+1),S,L)
will deliver the corresponding string of characters in S[1],...,S[L]. (S[L+1],...,S[STRINGMAX] are undefined.)

If X_i is neither NAME, KONST, nor STRING, then the call of GETSTRING will deliver an arbitrary string(usually the empty string).

5.2.3 Using the Semantic Stack

In order fully to understand this section, the user must have some knowledge of how an LR-parser works. The essence of this is given in part IV.

A stack is a useful tool when implementing the semantics of a language, especially if the language contains constructions, which may be nested. For this reason, the Skeleton-compiler contains a stack which operates in parallel with the parse stack:

ATTSTACK: ARRAY[STACKINX] OF ATTRIBUTES;
ATTRIBUTES = RECORD ... END;

The user may define fields in record ATTRIBUTES. A field in this record will be called an attribute, and the entire record an attribute record.

During parsing each symbol on the parse stack has a corresponding attribute record on ATTSTACK. When a reduction is performed and CODE is called, the topmost elements of ATTSTACK correspond to the symbols on the right side of the applied production.

In CODE the values of these attribute records of the right-side symbols may then be used in the semantic action. The semantic action of the applied production should define the value of the attribute record of the left-side symbol.

In order to access the relevant attribute records, CODE is supplied with two parameters:

OLDTOP,NEWTOP: STACKINX;

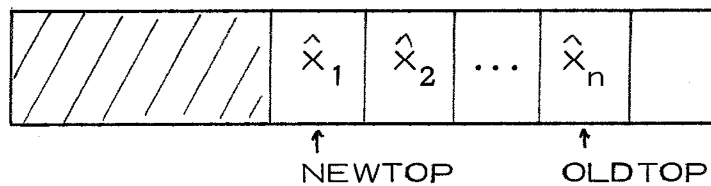
OLDTOP is the index in ATTSTACK of the topmost element before the reduction (at entry to CODE). NEWTOP is the index of the topmost element after the reduction is performed (after exit from CODE). If the right side of the applied production has the length N, then OLDTOP = NEWTOP+N-1. NEWTOP is furthermore the index of the attribute record which will correspond to the left side of the applied production. I.e. the attribute record of the first symbol on the right side will be used (after modifications done by the user) as the attribute record of the left side.

Let the applied production be

$$A \rightarrow X_1 X_2 \ldots X_n$$

At entry to CODE the situation is:

ATTSTACK:



$\hat{X}_i$ (i = 1, 2, ..., n) denotes the attribute record of X_i. Then

ATTSTACK[NEWTOP+i-1] or
ATTSTACK[OLDTOP-n+i]

is the attribute record of X_i. If i = 1, then

ATTSTACK[NEWTOP] or
ATTSTACK[OLDTOP-n+1]

is also the attribute record of A.

Lastly we would like to make some remarks concerning some relevant theoretical models for specifying semantics.

Attribute grammars [21], Syntax Directed Translation Schemes [20], and Attributed Translations [22] may all be implemented by using the ATTSTACK.

However only certain restricted classes of these models are efficient to implement.

The following models should be straightforward to implement:

- attribute grammars using only synthesized attributes [21]
- postfix simple syntax directed translation schemes [20]
- generalized syntax directed translation schemes with translation elements which are not only string-valued and with the string translation grammars being postfix and simple [20]
- attributed polish translation grammars using only synthesized attributes [22].

The latter two models are in principle identical and are a combination of the first two alternatives. Informally the semantic action of a production in these models is:

- on basis of the attribute records of the right-side symbols do:
  - output some symbols,
  - define the value of the attribute record of the left-side symbol.

Translation schemes which are not postfix (polish) and/or simple, may also be implemented. This requires that the attribute record contains pointers to the translation elements, which have to be built in the form of a tree; for a further study see Aho & Ullmann [20, chapter 9].

The implementation of inherited attributes or translations is inefficient in the BOBS-system. The only way to do it is to build the entire syntax tree and then perform the evaluation of the attributes.

However in practice it suffices to keep the inherited (or context) information (e.g. a symbol table) in global variables, and update this information appropriately.

We conclude this section by advocating the following model for specifying the semantics:

- make the grammar postfix, i.e. code only has to be outputted at the time of a reduction,
- define a suitable set of (synthesized) attributes for each nonterminal,
- define a set of global variables for collecting declarative information (i.e. a symbol table).

5.3 Error Recovery

The parser will take error action if the user specifies a string which is not in the language generated by the grammar. The error will be detected at the earliest possible point: that is, the part of the input string which has been read up till this point will constitute a correct prefix of some string in the

language where the next symbol read is not a valid continuation of the already read part. The symbol at which an error is detected will be marked with a ↑ and a number. The number refers to the error message table which is part of the output from the parser generator (see 2.1.4). The number indicates in the error message table a set of terminals that would have been valid continuations at that point of the input string. Note that the set of terminals does not always contain all valid continuations.

When an error has been detected, the parser tries to continue parsing. This can be done in the following way:

Let $A \to \alpha$ be a production. Assume that an error happens in a part of the input which later may reduce to α and then to A. We then have recognized part of α. Let $\alpha = \alpha' \alpha''$ where α' is recognized (α' may be empty). The parse stack then contains $\varphi \alpha' \delta$, for some φ and δ. This means that δ and some of the input symbols (if there were no errors) could reduce to α'' and then $\alpha'\alpha''$ to A. A possible way of recovering would be to assume that α'' has been recognized. This can approximately be done by deleting δ from the stack and skip symbols on input until meeting one which may follow α''.

In practice the parser is simultaneously looking for several right sides of different productions which may reduce to different nonterminals. For this reason the user must specify the so-called error productions in order to indicate the nonterminals that the recovery algorithm should try to reduce to.

An error production has the form

$$A \to \alpha \text{ error } \beta$$

where error is a special terminal (see Section 4.2.2), and α and β are (possibly empty strings) of nonterminals and terminals.

Let $A \to \alpha\gamma$ be another A-production. Let the stack at a given time during the parse contain $\varphi \alpha \delta$ for some φ and δ. Assume that δ and some terminal string x may reduce to γ and then $\alpha\gamma$ to A. Assume that x is a prefix of some terminal string which is a valid continuation on input with the above stack.

Suppose that an error appears in this situation, i.e. the next input symbol is not a valid continuation. The parser will then replace γ by error β in the

following way:

- δ is deleted from the stack
- the symbol error is read
- input symbols are skipped until meeting one which may begin β.
  If β is the empty string one skips to a symbol which may follow
  A after α error is reduced to A.

As $A \to \alpha$ error β is a usual production. CODE will be called with the number of the error production. In this way the user is informed when syntax errors appear.

In general more than one error production may be applicable. In this case the parser will choose the one which gives rise to fewest skips on input.

As error productions are normal productions they may give rise to LR-conflicts which of course must be eliminated.

In order to obtain a successful recovery the user must specify a reasonable set of error productions. All parts of the grammar should be covered as should different levels in nested constructions.

In general, an error production $A \to \alpha$ error β only makes sense if α is a prefix of some other production with A as leftside, and β should in a similar way be a postfix. It is often sufficient to let α and/or β be the empty string.

5.4 Examples

Here is an example of using the skeleton-compiler and tables as produced by the parser generator in example 4.1.

Example 5.1.

```
(*BOBS EXAMPLE*)
DECLARE A,B:
   A:=1; B:=0;
   WHILE A <> (*THIS IS A COMMENT*) 10
   DO
      A:=A+1; B:=B++A;
   OD;
   WRITE(B).
```

Output from the Skeleton-compiler. Note, a syntax error is marked.

```
    (*BOBS EXAMPLE*)
  DECLARE A,B;
    A:=1; B:=0;
    WHILE A <> (*THIS IS A COMMENT*) 10
    DO
      A:=A+1; B:=B++A;
                   ^ 4
    OD;
    WRITE(B).
```

```
SNAPSHOT:

 LEXICAL: (*
 LEXICAL: DECLARE
 LEXICAL: A
 LEXICAL: ,
 PRODUCTION:  9
   SYMB1 A
 PRODUCTION:  8
 LEXICAL: B
 LEXICAL: ;
 PRODUCTION:  9
   SYMB1 B
 PRODUCTION:  7
 LEXICAL: A
 PRODUCTION:  3
 LEXICAL: :=
 PRODUCTION: 23
   SYMB1 A
 LEXICAL: 1
 LEXICAL: ;
 PRODUCTION: 11
   SYMB1 1
 PRODUCTION: 41
 PRODUCTION: 37
 PRODUCTION: 31
 PRODUCTION: 26
 PRODUCTION: 15
 PRODUCTION:  6
 LEXICAL: B
 LEXICAL: :=
 PRODUCTION: 23
   SYMB1 B
 LEXICAL: 0
 LEXICAL: ;
 PRODUCTION: 11
   SYMB1 0
 PRODUCTION: 41
 PRODUCTION: 37
 PRODUCTION: 31
 PRODUCTION: 26
 PRODUCTION: 15
 PRODUCTION:  5
 LEXICAL: WHILE
 LEXICAL: A
 LEXICAL: <>
 PRODUCTION: 23
   SYMB1 A
 PRODUCTION: 40
 PRODUCTION: 37
 PRODUCTION: 31
 LEXICAL: (*
 LEXICAL: 10
 PRODUCTION: 33
 LEXICAL: DO
 PRODUCTION: 11
   SYMB1 10
 PRODUCTION: 41
 PRODUCTION: 37
 PRODUCTION: 31
 PRODUCTION: 25
 LEXICAL: A
 LEXICAL: :=
 PRODUCTION: 23
   SYMB1 A
 LEXICAL: A
 LEXICAL: +
 PRODUCTION: 23
   SYMB1 A
 PRODUCTION: 40
 PRODUCTION: 37
 PRODUCTION: 31
 LEXICAL: 1
 PRODUCTION: 34
 LEXICAL: ;
 PRODUCTION: 11
   SYMB1 1
 PRODUCTION: 41
 PRODUCTION: 37
 PRODUCTION: 30
 PRODUCTION: 26
 PRODUCTION: 15
 PRODUCTION:  6
 LEXICAL: B
 LEXICAL: :=
 PRODUCTION: 23
   SYMB1 B
 LEXICAL: B
 LEXICAL: +
 PRODUCTION: 23
   SYMB1 B
 PRODUCTION: 40
 PRODUCTION: 37
 PRODUCTION: 31
 LEXICAL: +
 PRODUCTION: 34
 LEXICAL: A <---SYNTAXERROR <---SKIPPED
 LEXICAL: ;
 PRODUCTION: 27
 PRODUCTION: 15
 PRODUCTION:  5
 LEXICAL: OD
 PRODUCTION: 14
 PRODUCTION:  5
 LEXICAL: ;
 PRODUCTION: 17
 PRODUCTION:  5
 LEXICAL: WRITE
 LEXICAL: (
 LEXICAL: B
 LEXICAL: )
 PRODUCTION: 23
   SYMB1 B
 PRODUCTION: 40
 PRODUCTION: 37
 PRODUCTION: 31
 PRODUCTION: 26
 PRODUCTION: 28
 LEXICAL: .
 PRODUCTION: 20
 PRODUCTION:  5
 LEXICAL:
 PRODUCTION:  1
 LEXICAL:
```

Example 5.2

This example shows how a simple translation can be implemented. We define a small language with simple control structures, assignments, expressions and Algol like blocks with declaration of variables. A variable must be declared before it is used and double declarations in a block may not appear. All language constructs are translated into an equivalent one. The major transformations are : variables are added the nesting depth of the block in which they are declared, expressions are transformed to postfix polish, the control structures are extended with line numbers indicating possible jumps. The grammar has been transformed in order to make the translation polish.

The following pages contain

- a listing of the grammar,
- all global label, const, type, and var declarations of the skeleton compiler, including the ones added for the semantic example,
- the part of the procedure CODE which has been added for the semantic example, and
- an example of a translation.

```
****************************** THE GRAMMAR *********************

 1   <PROGRAM> ::= <START> <BLOCK> .

 2   <START> ::= EMPTY

 3   <BLOCK> ::= <BLOCKDEC> <STATEMENTSEQ> END

 4   <BLOCKDEC> ::= <BEGIN> <DECLARATION>

 5   <STATEMENTSEQ> ::= <STATEMENTSEQ> ; <STATEMENT>
 6                    / <STATEMENT>

 7   <BEGIN> ::= BEGIN

 8   <DECLARATION> ::= <DECID> ; <DECLARATION>
 9                   / EMPTY

10   <DECID> ::= DECLARE NAME

11   <STATEMENT> ::= <IFTHEN> <STATEMENTSEQ> FI
12                 / <WHILECLAUSE> <STATEMENTSEQ> OD
13                 / <BLOCK>
14                 / <VAR> := <EXP>

15   <IFTHEN> ::= <IFCLAUSE> <STATEMENTSEQ> ELSE

16   <WHILECLAUSE> ::= <WHILE> <EXP> DO

17   <VAR> ::= NAME

18   <EXP> ::= <EXP> + <TERM>
19           / <TERM>

20   <IFCLAUSE> ::= <IF> <EXP> THEN

21   <IF> ::= IF

22   <WHILE> ::= WHILE

23   <TERM> ::= <TERM> * <PRIMARY>
24            / <PRIMARY>

25   <PRIMARY> ::= KONST
26               / <VAR>
27               / ( <EXP> )

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

```
000006
000006     (*                          *)
000006     (*  B O B S - SYSTEM        *)
000006     (*                          *)
000006     (*  SKELETON COMPILER       *)
000006     (*                          *)
000006     (*  VERSION MARCH 1977      *)
000006
000006  PROGRAM BOBS(INPUT,OUTPUT,TABLES);
000464
000464 LABEL 10; (*EXIT LABEL*)
000464 CONST
000464    STACKMAX=50; (*SIZE OF ATTSTACK AND PARSESTACK *)
000464    STRINGMAX=100; (* SIZE OF ATTRIBUTE STRING *)
000464    CHBUFMAX=200; (* SIZE OF ARRAY CHBUF *)
000464    MINCH='A'; MAXCH=';'; (*FIRST/LAST CHARACTER IN TYPE CHAR*)
000464    TEST=TRUE; (* IF TRUE THEN SNAPSHOTS ARE GENERATED*)
000464    (* CONSTANT DEFINITIONS OF THE USER *)
000464    CODEMAX=100;
000464    BNMAX=10;
000464    SYMBMAX=100;
000464 TYPE
000464    CHBUFINX=0..CHBUFMAX;
000464    STACKINX=0..STACKMAX;
000464    STRING=PACKED ARRAY[1..STRINGMAX] OF CHAR;
000464    (* TYPE DEFINITIONS THE USER *)
000464    ATTRIBUTES=RECORD
000464                   CHBUFP:  CHBUFINX; (*USED BY PROCEDURE GETSTRING *)
000464                   REMEMBER,WHILESTART:INTEGER
000464                END;
000464    OPKIND=0..2;
000464    OPERATION=RECORD
000464                  OP:ALFA;
000464                  CASE K:OPKIND OF
000464                    1:(ARG:INTEGER);
000464                    2:(ID:STRING;BN:0..BNMAX)
000464                   END;
000464 VAR
000464    ATTSTACK: ARRAY[STACKINX] OF ATTRIBUTES;
000715    CHBUF: ARRAY[CHBUFINX] OF CHAR;
001226    CHBUFI: CHBUFINX;
001227    OK: BOOLEAN;
001230    TABLES: TEXT;
001457    SNAPSHOTS: TEXT;
001706    (*CHBUF, CHBUFI, FIELD CHBUFP OF ATTRIBUTES, TABLES AND OK
001706    (*SHOULD NOT BE CHANGED BY THE USER *)
001706    (* VAR DECLARATIONS OF THE USER *)
001706    DISPLAY: ARRAY[0..BNMAX] OF INTEGER;
001721    BLOCKNO: INTEGER;
001722    SYMBOLTABLE: ARRAY[0..SYMBMAX] OF
001722                    RECORD
001722                      IDENT: STRING;
001722                      LEVEL:INTEGER
001722                    END;
004051    SYMBNO: INTEGER;
004052    CODEARRAY:ARRAY[1..CODEMAX] OF OPERATION;
006476    CODEINX:0..CODEMAX;
006477
006477   (*$L-*)
000004 (*$L+*)
```

```
000004 (*$F*)
000004 PROCEDURE CODE(OLDTOP,NEWTOP: STACKINX; PROD: INTEGER);
000006  VAR STR: STRING;
000020     I,L,V:INTEGER;
000023
000023     PROCEDURE PRINTOUT;
000003     VAR I: INTEGER;
000004     BEGIN WRITELN(OUTPUT);
000007        FOR I:=1 TO CODEINX DO
000011          WITH CODEARRAY[I] DO
000020            BEGIN WRITE(I:5,'    ',OP);
000036              CASE K OF
000043                0: WRITELN;
000045                1: WRITELN(ARG:3);
000054                2: WRITELN(ID:10,BN:3)
000066              END
000073            END
000073     END;
000103
000103     PROCEDURE GEN0(OPCODE:ALFA);
000004     BEGIN CODEINX:=CODEINX+1;
000012       WITH CODEARRAY[CODEINX] DO
000017       BEGIN
000017         OP:=OPCODE; K:=0;
000020       END;
000020     END;
000023
000023     PROCEDURE GEN1(OPCODE:ALFA;ARGUMENT:INTEGER);
000005     BEGIN CODEINX:=CODEINX+1;
000013       WITH CODEARRAY[CODEINX] DO
000017       BEGIN
000017         OP:=OPCODE; K:=1;
000020         ARG:=ARGUMENT;
000021       END;
000021     END;
000026
000026     PROCEDURE GEN2(OPCODE: ALFA;IDF:STRING;LEVEL:INTEGER);
000020     BEGIN CODEINX:=CODEINX+1;
000017       WITH CODEARRAY[CODEINX] DO
000023       BEGIN
000023         OP:=OPCODE; K:=2;
000024         ID:=IDF; BN:=LEVEL;
000033       END;
000033     END;
000040
000040     PROCEDURE GENBACKWARD(OPCODE:ALFA;INMIND:BOOLEAN);
000005     BEGIN
000005       WITH ATTSTACK[NEWTOP] DO
000014       BEGIN
000014         WITH CODEARRAY[REMEMBER] DO
000021         BEGIN K:=1;
000022           OP:=OPCODE;
000022           ARG:=CODEINX+1;
000024         END;
000024         IF INMIND THEN
000024         BEGIN
000024           CODEINX:=CODEINX+1;
000027           REMEMBER:=CODEINX;
000027         END;
000027       END;
```

```
000027      END;
000034
000034      PROCEDURE DECLAREID(ID: STRING);
000016      VAR I:INTEGER;
000017      BEGIN
000017        I:=DISPLAY[BLOCKNO-1]+1;
000017        WHILE (ID<>SYMBOLTABLE[I].IDENT) AND (I<SYMBNO) DO I:=I+1;
000034        IF (ID=SYMBOLTABLE[I].IDENT) AND (I<=SYMBNO)
000046        THEN WRITELN(' ',ID,' ALREADY DECLARED')
000063        ELSE
000065         BEGIN SYMBNO:=SYMBNO+1;
000066            SYMBOLTABLE[SYMBNO].IDENT:=ID;
000075            SYMBOLTABLE[SYMBNO].LEVEL:=BLOCKNO;
000102            GEN2('DECLARE   ',ID,BLOCKNO);
000105         END;
000105      END;
000115
000115      PROCEDURE USEID(ID:STRING);
000016      VAR I,J:INTEGER;
000020      BEGIN I:=SYMBNO;
000014         WHILE (ID<>SYMBOLTABLE[I].IDENT) AND (I>0) DO I:=I-1;
000031         WITH SYMBOLTABLE[I] DO
000035            IF ID=IDENT THEN GEN2('VAR       ',ID,LEVEL)
000044            ELSE WRITELN(OUTPUT,' ',ID,' IS NOT DECLARED');
000064      END;
000076
000076   (*$L-*)
```

```
RECAU PASCAL VER.  2.2/1-28            77/06/01. 13.29.51.

000147    (*$L+*)(*$F*)
000147      BEGIN (*CODE*)
000147
000147        CASE PROD OF
000014        0: (* START PRODUCTION ADDED BY BOBS *) ;
000015        1: (* <PROGRAM>::=<START> <BLOCK> . *)
000015           BEGIN
000015             GEN0('ENDPROGRAM');
000017             PRINTOUT;
000021           END;
000022        2: (* <START> ::= EMPTY *)
000022           BEGIN
000022             BLOCKNO:=0;
000023             SYMBNO:=0;
000024             CODEINX:=0;
000025             GEN0('PROGRAM    ');
000027           END;
000030        3: (* <BLOCK> ::= <BLOCKDEC> <STATEMENTSEQ> END *)
000030           BEGIN
000030             GEN0('END        ');
000032             BLOCKNO:=BLOCKNO-1;
000034             SYMBNO:=DISPLAY[BLOCKNO];
000037           END;
000040        4: (* <BLOCKDEC> ::= <BEGIN> <DECLARATION> *)
000040             GEN0('CODE       ');
000043        5: (* <STATEMENTSEQ> ::= <STATEMENTSEQ> ; <STATEMENT> *);
000044        6: (* <STATEMENTSEQ> ::= <STATEMENT> *);
000045        7: (* <BEGIN> ::= BEGIN *)
000045           BEGIN
000045             GEN0('BLOCK      ');
000047             DISPLAY[BLOCKNO]:=SYMBNO;
000053             BLOCKNO:=BLOCKNO+1;
000054           END;
000055        8: (* <DECLARATION> ::= <DECID> ; <DECLARATION> *);
000056        9: (* <DECLARATION> ::= EMPTY *);
000057        10: (* <DECID> ::= DECLARE NAME *)
000057            BEGIN
000057              GETSTRING(2,STR,L);
000062              FOR I:=L+1 TO STRINGMAX DO STR[I]:=' ';
000102              DECLAREID(STR);
000104            END;
000105        11: (* <STATEMENT> ::= <IFTHEN> <STATEMENTSEQ> FI *)
000105            BEGIN
000105              GENBACKWARD('ELSE       ',FALSE);
000112              GEN0('FI         ');
000114            END;
000115        12: (* <STATEMENT> ::= <WHILECLAUSE> <STATEMENTSEQ> OD *)
000115            BEGIN
000115              GENBACKWARD('DO         ',FALSE);
000122              GEN1('OD         ',ATTSTACK[NEWTOP].WHILESTART);
000131            END;
000132        13: (* <STATEMENT> ::= <BLOCK> *);
```

```
000133        (*$F*)
000133        14: (* <STATEMENT> ::= <VAR> := <EXP> *)
000133            GEN0('STORE     ');
000136        15: (* <IFTHEN> ::= <IFCLAUSE> <STATEMENTSEQ> ELSE *)
000136              GENBACKWARD('THEN      ',TRUE);
000143        16: (* <WHILECLAUSE> ::= <WHILE> <EXP> DO *)
000143              GENBACKWARD('WHILE     ',TRUE);
000150        17: (* <VAR> ::= NAME *)
000150            BEGIN
000150              GETSTRING(1,STR,L);
000153              FOR I:=L+1 TO STRINGMAX DO STR[I]:=' ';
000173              USEID(STR);
000175            END;
000176        18: (* <EXP> ::= <EXP> + <TERM> *)
000176            GEN0('PLUS      ');
000201        19: (* <EXP> ::= <TERM> *);
000202        20: (* <IFCLAUSE> ::= <IF> <EXP> THEN *)
000202              GENBACKWARD('IF        ',TRUE);
000207        21: (* <IF> ::= IF *)
000207            BEGIN CODEINX:=CODEINX+1;
000213              ATTSTACK[NEWTOP].REMEMBER:=CODEINX;
000217            END;
000217        22: (* <WHILE> ::= WHILE *)
000217            BEGIN CODEINX:=CODEINX+1;
000223              WITH ATTSTACK[NEWTOP] DO
000227               BEGIN
000227                  WHILESTART:=CODEINX;
000230                  REMEMBER:=CODEINX;
000230               END;
000230           END;
000231        23: (* <TERM> ::= <TERM> * <PRIMARY> *)
000231             GEN0('MULT      ');
000234        24: (* <TERM> ::= <PRIMARY> *);
000235        25: (* <PRIMARY> ::= KONST *)
000235             BEGIN
000235              GETSTRING(1,STR,L);
000240              V:=0;
000241              FOR I:=1 TO L DO V:=V*10+ORD(STR[I])-ORD('0');
000260              GEN1('LIT       ',V);
000263            END;
000264        26: (* <PRIMARY> ::= <VAR> *)
000264            GEN0('LOAD      ');
000267      27: (* <PRIMARY> ::= ( <EXP> ) *);
000270        END;
000324      END; (*CODE*)
000364
000364   (*$L-*)
```

```
BEGIN
   DECLARE A; DECLARE B;
   A:=10; B:=A;
   IF A+B THEN
     BEGIN DECLARE A;
        A:=B
     END
   ELSE
     BEGIN DECLARE B;
        WHILE A DO B:=A+1 OD
     END
   FI
END .
```

```
 1   PROGRAM
 2   BLOCK
 3   DECLARE    A           1
 4   DECLARE    B           1
 5   CODE
 6   VAR        A           1
 7   LIT         10
 8   STORE
 9   VAR        B           1
10   VAR        A           1
11   LOAD
12   STORE
13   IF          19
14   VAR        A           1
15   LOAD
16   VAR        B           1
17   LOAD
18   PLUS
19   THEN        28
20   BLOCK
21   DECLARE    A           2
22   CODE
23   VAR        A           2
24   VAR        B           1
25   LOAD
26   STORE
27   END
28   ELSE        44
29   BLOCK
30   DECLARE    B           2
31   CODE
32   WHILE       35
33   VAR        A           1
34   LOAD
35   DO          42
36   VAR        B           2
37   VAR        A           1
38   LOAD
39   LIT          1
40   PLUS
41   STORE
42   OD          32
43   END
44   FI
45   END
46   ENDPROGRAM
```

PART III

6. Program Description

In this section the program organization is outlined and algorithms and data structures of special interest are described in detail.

The description is not of any interest to the ordinary user of the system and may be skipped. No important information for the normal use of the system is contained in this part.

The present description is addressed to users (or just readers) who are specially interested in a detailed description of:

- the program organization
- the methods behind the algorithms, or
- the actual appearance of the algorithms.

The reader is assumed to be familiar with the terminology and the concepts in consideration are assumed to be well known to the reader. References [1, 2, 3, 4, 5, 18, 19, 20] may be helpful to provide the necessary background.

The description consists of two parts:

- a description of the parser-generator,
- a description of the skeleton compiler.

6.1 Description of the Parser-Generator

The parser-generator program may be described as three separate modules:

- the input module,
  which transforms an external grammar specification into an internal representation,
- the grammar transformation module,
  which consists of a set of routines, each performing either a grammar check or a grammar transformation - or both,

- the parse table construction module
  which builds the parse tables according to the specified grammar. The parse tables are organized as a directed cyclic graph where the nodes are linked lists. The information part of a given list is either a sequence of grammar symbols or a sequence of references to graph nodes.

6.1.1 The Construction of the Parse Tables

The construction of the parse tables consists of a sequence of steps:

- the construction of the LR(0) tables; (which is a realization of the method described in [2]);

- the extension to LALR(1)-tables which supplies any inadequate table among the LR(0)-tables with LALR(1)-lookahead information, which may solve the existing conflict. A table supplied in this way is called a lookahead table.
  SLR(1) lookahead information may be sufficient to solve a conflict and may then be used;

- the construction of special lookback tables, defined in [1]. The tables are constructed on the basis of the grammar and the LR(0) tables: one lookback table for each nonterminal symbol. This construction method deviates from the one given in [1] but includes automatically most of the optimizations of the lookback tables, also given in [1];

- the optimization of the lookback tables, described in [1, 4], which involves the addition of an "In Any Case" condition for the most popular destination table in a lookback table;

- the optimization of the parse tables which removes the information concerning the transitions on nonterminal symbols, given in [1]. When the lookback tables are added to the parse tables this information becomes redundant.

- The optimization of the lookahead tables, first described in [4], which involves the addition of an "In Any Case" condition for

the most popular destination table in a lookahead table;

- the optimization of (unaltered) LR(0) tables which merges tables having identical bottom-most subsets. The tables are modified to share the common part. This optimization may be regarded as a variant of the method using top-most subsets, given in [4].

6.1.2 The LALR(1) Lookahead Algorithm

A description of the LALR(1) algorithm may be found in [23].

6.2 Description of the Skeleton Compiler

This program consists logically of two parts:

- The semantic interface,

  which consists of a list of predefined structures and auxiliary routines which may be useful to the user. Furthermore these define the means by which the user and the parser may interact, (see section 5.2)

- The parser,

  which primarily consists of a parsing algorithm working on the parse tables constructed by the parser generator program. This parsing algorithm is described in the next section.

  The lexical analyzer is an independent part of the parser. It runs on tables which are also built up by the parser generator program.

  The parser includes an error recovery routine based on the parse tables. This routine is described in detail in section 6.2.2.

6.2.1 The Parsing Algorithm

In addition to the parse tables, the parsing algorithm uses a parse stack. A picture is a special parse situation which we may define as follows:

Assume that the parse tables are constructed on the basis of a CFG, $G = (N, \Sigma, P, S)$. Let t_k be a parse table, ($k \in [0, i]$), and $X_k \in N \cup \Sigma$, ($k \in [1, i-1]$), and $X_i = a_j$, and $a_k \in \Sigma$, ($k \in [1, n]$) and $1 \leq j \leq n$.

The general picture, P_j, is then:

$$\boxed{t_0 X_1 t_1 X_2 t_2 \ldots X_i t_i} \qquad a_{j+1} a_{j+2} \ldots a_n$$

parse stack

The initial picture, P_0, is:

$$\boxed{t_0} \qquad a_1 a_2 \ldots a_n$$

and the final picture, P_n, is:

t_0 S

The picture P_j, $0 < j$, denotes the situation where the symbol a_j (and a parse table t_i) has just been shifted into the stack. The parse tables t_{i-1} and t_i which are also on the stack mean that there exists a transition path on the symbol a_j from table t_{i-1} to table t_i. This fact may be generalized to any sequence on the stack of the form $t_{k-1}X_k t_k$.

Assuming that the string $a_1 a_2 \dots a_n$ is in L(G) the parsing algorithm runs through the sequence of pictures

$$P_0, P_1, \dots, P_n .$$

Between any two pictures in this sequence the stack may increase and decrease according to intermediate parse situations. The parsing algorithm may be specified as:

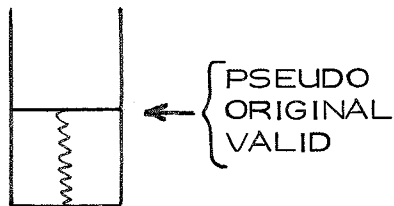
```
P := P0;
WHILE input not exhausted DO
BEGIN
      IF ¬ lookahead (P) THEN recover (P);
      P := successor (P);
END;
```

The routines involved behave as follows:

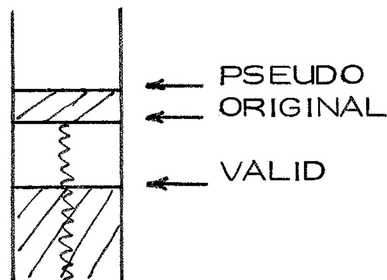
- Lookahead,

determines if there exists a picture P_{i+1} as a successor to P_i. The calculation process may involve a sequence of both reductions of the parse stack and shifts onto the parse stack. In this context a shift is due to a reduction of an empty production. P_i is not touched if the stack is extended into an area on top of the original stack. The differences between the original picture and the succeeding picture are contained in this area.

The picture P_i may be depicted as follows:

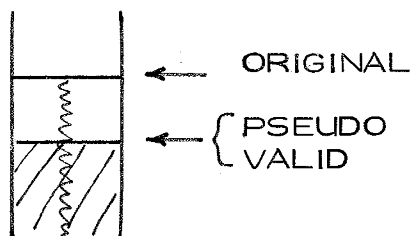


The three indices into the stack all indicate the actual top of the stack. In the lookahead routine the stack may change as described above. Thus the stack situation during the computation may appear as:



As stated the original picture P_i must be protected. The area in question is indicated by the index ORIGINAL, which remains fixed during the look-ahead computation.

The hatched area of the above illustration shows the part of the stack which is being used in the computations at this moment. The area situated between ORIGINAL and PSEUDO contains a part of stack which might have been placed in the unhatched area upwards from VALID (thus causing the destruction of the original stack). The special case (which in fact is the common case) when this area between ORIGINAL and PSEUDO is empty, is perceived according to the following illustration:



The area (in both the latter figures) below VALID is the part of the original stack which is still in use at the moment.

We note that the index PSEUDO may both increase and decrease, whereas VALID may only decrease.

When the lookahead routine terminates it leaves a stack situation similar to one of the illustrations.

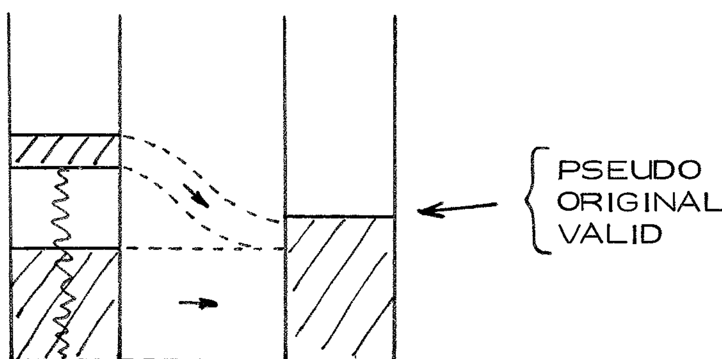
During the lookahead computation adequate information about the possible sequence of reductions (mentioned above) is queued up for use in the successor routine.

- Successor,

delivers the information about the possible reduction sequence to the semantic interface part of the program.

On the basis of the extra information in the area on the top of the parse stack a succeeding stack is built, resulting in the picture P_{i+1}.

The transformation of the intermediate stack to the succeeding picture may be depicted:



(The transformation of the alternative intermediate stack situation is trivial.) All three indices are reset to the resulting top of the stack.

- Recover,

transforms the picture P_i into another picture P'_k, such that Lookahead (P'_k) is TRUE. Recover involves the construction of another stack and the deletion of a number of input symbols. The routine is described in detail in the next section.

A somewhat different description of the parsing principle applied above is given in [19].

6.2.2 The recovery Algorithm

The recovery algorithm is invoked when parsing a string not in L(G). However having detected an error several possibilities of reaction exist. A safe but normally insufficient reaction is to stop the parsing and inform the user about the discovered error. A tempting reaction is to try to correct the input string, i.e. to transform the erroneous input string into an error-free string. However in general the correction scheme is extremely complicated. An appealing compromise is the recovery principle in which the parser establishes a situation such that a major part of the remaining input string may also be inspected to discover any additional errors. The process may involve both the restructuring of the parse stack and the deletion of symbols from the input string.

6.2.2.1 Description of the Algorithm

The recovery algorithm is activated from the parsing algorithm when lookahead (P_j) is FALSE for some j. Let P_j be

$$\boxed{t_0 X_1 t_1 \ldots X_i t_i} \qquad a_{j+1} a_{j+2} \ldots a_n$$

The recovery method complies with an outline given in [18] and works by applying error productions. An error production has the form $A \to \alpha$ error β where error is a special symbol.

In the algorithm the remaining input symbols are inspected one symbol at a time. For each symbol a sequence of stack situations is inspected, which is constructed from the original stack by popping one stack element at a time. For each such stack situation the special error symbol is inserted as the first symbol on input. This artificial picture is now inspected. If a successor to this picture exists, to which, furthermore, another successor exists then a straightforward way of recovering from the error situation has appeared. (Involving the described deletion of input symbols and popping of stack elements.)

The method may be formalized in the following algorithm: (based on P_j)

```
FOR k := j + 1 TO n DO
    FOR l := i DOWNTO 0 DO
    BEGIN
        LET P^{l,k}_error
        BE [ t_0 X_1 t_1 ... X_l t_l   error a_k ... a_n
        IF lookahead (P^{l,k}_error) THEN
        BEGIN
            P'_k := successor (P^{l,k}_error);
(*)         IF lookahead (P'_k) THEN
            BEGIN
                delete symbols a_j, a_{j+1}, ..., a_{k-1};
                EXIT TO success;
            END;
        END;
    END;
    success: (*successful picture: P'_k *)
```

Note: A straightforward improvement of the algorithm is to extend the check performed at (*) to include 2-3 symbols instead of only a single symbol.

6.2.2.2 Description of the Recovery Principle

The detection of an error is caused by a picture P_j: $\boxed{t_0 \ldots X_i t_i}$ $a_{j+1} \ldots a_n$, where $X_i = a_j$.

The existence of P_j implies $\exists\, y \in \Sigma^*$ such that

$$S \Rightarrow^*_{rm} X_1 \ldots X_i\, y \qquad (1)$$
$$\Rightarrow^*_{rm} a_1 \ldots a_j\, y$$

The recovery method involves a specification of k and l ($j < k \leq n$, $0 \leq l \leq i$) and thereby the picture

$P^{l,k}_{error}$: $t_0 \ldots X_l t_l$ error $a_k \ldots a_n$ with the property that

$$\text{lookahead}\ (P^{l,k}_{error}) = \text{TRUE} \qquad (2)$$

and assuming that

$$P'_k := \text{successor}\ (P^{l,k}_{error})$$

the property that

$$\text{lookahead}\ (P'_k) = \text{TRUE}. \qquad (3)$$

Inspecting the method step by step a sequence of conclusions may be stated based on the results above:

(1) implies that for a given $l \leq i$, $\exists\, s(0 \leq s \leq j)$ and $\exists\, z \in \Sigma^*$ such that

$$S \Rightarrow^*_{rm} X_1 \ldots X_l X_{l+1} \ldots X_i\ y$$
$$\Rightarrow^*_{rm} X_1 \ldots X_l\ zy$$
$$\Rightarrow^*_{rm} a_1 \ldots a_s\ zy$$

(2) may be interpreted to mean that there exist $x = x'' x' \in \Sigma^*$, m ($0 \leq m \leq l$) and an error production $A \rightarrow \alpha$ error β such that

$$S \Rightarrow^*_{rm} X_1 \ldots X_m\ A\ x'$$
$$\Rightarrow_{rm} X_1 \ldots X_m X_{m+1} \ldots X_l \text{ error } \beta\ x'$$
$$\Rightarrow^*_{rm} X_1 \ldots X_l \text{ error } x''\ x'$$
$$\Rightarrow^*_{rm} a_1 \ldots a_s \text{ error } x$$

where $\beta \Rightarrow^*_{rm} x''$ and $\alpha = X_{m+1} \ldots X_l$.

(3) implies that at least one of the potential error productions, say $A \rightarrow \alpha$ error β, is legal when the symbol a_k is included. Hence $\exists\, w = w''\ w' = a_k\ v$ such that

$$S \Rightarrow^*_{rm} X_1 \ldots X_m\ A\ w'$$
$$\Rightarrow_{rm} X_1 \ldots X_l \text{ error } \beta\ w'$$
$$\Rightarrow^*_{rm} X_1 \ldots X_l \text{ error } w''\ w'$$
$$\Rightarrow^*_{rm} a_1 \ldots a_s \text{ error } a_k\ v$$

where $\beta \Rightarrow^*_{rm} w''$ and $\alpha = X_{m+1} \ldots X_l$.

We may now summarize the results. The parsing algorithm has gone through the parse situations

$$\boxed{{}_{t_0}X_1{}_{t_1} \cdots {}_{t_l}X_l} \qquad a_{s+1} \cdots a_j \cdots a_k \cdots a_n$$

and

$$\boxed{{}_{t_0}X_1{}_{t_1} \cdots {}_{t_l}X_l \cdots {}_{t_i}X_i} \qquad a_{j+1} \cdots a_k \cdots a_n$$

at which the error is detected.

This situation is transformed into

$$\boxed{{}_{t_0}X_1{}_{t_1} \cdots {}_{t_l}X_l} \qquad \text{error } a_k \cdots a_n$$

The effect of the recovery method is to discard the symbols $X_{l+1} \cdots X_i$ from the stack and to delete the symbols $a_{j+1} \cdots a_{k-1}$ from the input string. Recalling that

$$X_{l+1} \cdots X_i \Rightarrow^*_{rm} a_{s+1} \cdots a_j$$

the total effect is to discard the symbol string $a_{s+1} \cdots a_{k-1}$. The string is replaced by the special symbol error which is part of a string derivable from the language construct A.

Now the philosophy behind the recovery method may be expressed as: While parsing the string $a_1 \cdots a_n$ an error is detected. The error is located in the partial string $a_{s+1} \cdots a_{k-1}$ which is supposed to reduce to a part of the language construct A. A part of the string has been reduced into $X_{l+1} \cdots X_i$ and a part remains on input. Both these parts are discarded. The succeeding picture $P'_k : \boxed{t_0 \cdots X_r\, t_r} \quad a_k \cdots a_n$ may be interpreted as an element of the picture sequence evaluated from the initial picture

$$P'_0 : \boxed{t_0} \qquad a_1 \cdots a_s \text{ error } a_k \cdots a_n.$$

Pictures among the successors of the picture P'_k may cause new error detections.

PART IV

7. Summary of terminology

In this section we shall give a short summary of the notion of a context-free grammar, some parsing terminology, and an introduction to LR-parser construction. For a more detailed and explanatory treatment see Aho & Ullman [20].

Let M be a set of symbols, then M^* denotes the set of all strings of symbols from M, furthermore it includes the empty string (denoted by e).

Example 7.1

Let $A = \{0, 1\}$, then

$$A^* = \{e, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$$

□

7.1 Context free grammar

A context-free grammar (CFG) is a 4-tuple $G = (N, \Sigma, P, S)$, where

(1) N is a finite set of nonterminal symbols.

(2) Σ is a finite set of terminal symbols.

(3) P is a finite set of productions each on the form:

$$A \rightarrow \alpha$$

where A is in N and α is in $(N \cup \Sigma)^*$.

(4) S is a distinguished symbol in N called the start symbol (or goal symbol). □

A sentential form of a grammar G is defined recursively as follows:

(1) S is a sentential form.

(2) If $\alpha B \gamma$ is a sentential form, and $B \rightarrow \delta$ is a production in P, then $\alpha \delta \gamma$ is a sentential form. We say that $\alpha B \gamma$ directly derives $\alpha \delta \gamma$ and denote this by:

$$\alpha B \gamma \Rightarrow \alpha \delta \gamma$$

□

A sentential form containing no nonterminals is called a <u>sentence</u> generated by G.

The <u>language generated by a grammar</u> G, denoted L(G), is the set of sentences generated by G. I.e. a grammar G defines a language of strings of its terminal symbols. □

We use the following conventions

$\alpha, \beta, \gamma, \delta, \ldots$	are in $(N \cup \Sigma)^*$,
a, b, c, d, ...	are in Σ
v, x, y, z, w, ...	are in Σ^*
A, B, C, ...	are in N

Let α, β be sentential forms. We say that α <u>derives</u> β (and denote it by $\alpha \Rightarrow^* \beta$ if and only if:

$\alpha = \beta$ or
there exist sentential forms
$\gamma_0, \gamma_1, \ldots, \gamma_n$, $n \geq 1$ and
$\alpha = \gamma_0$, $\beta = \gamma_n$, such that
$\gamma_0 \Rightarrow \gamma_1$,
$\gamma_1 \Rightarrow \gamma_2$,
⋮
$\gamma_{n-1} \Rightarrow \gamma_n$.

The sequence:

$$\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \gamma_{n-1} \Rightarrow \gamma_n$$

is called a <u>derivation</u> of β from α.

A derivation is called a <u>rightmost derivation</u> if, in each step $\gamma_i \Rightarrow \gamma_{i+1}$, it is always the rightmost nonterminal in γ_i which is replaced by means of a production. The symbols $\Rightarrow_{rm}$ $\Rightarrow^*_{rm}$, indicate rightmost derivations.

A nonterminal E is called <u>left recursive</u> if and only if $E \Rightarrow \alpha \Rightarrow^* E \alpha'$ for some α, α'. <u>Right recursive</u> is defined in a similar way.

□

A grammar G is <u>ambiguous</u> if and only if some sentence w in L(G) has more than one distinct rightmost derivation. □

Example 7.2

Consider the grammar

$$G_0 = (\{E, T, P\}, \{+, *, (,), a\}, \mathfrak{P}, E),$$

where $\mathfrak{P}$ consists of

$E \to E + T$

$E \to T$

$T \to T * P$

$T \to P$

$P \to (E)$

$P \to a$

An example of a rightmost derivation in G_0 is:

$$\begin{aligned} E &\Rightarrow_{rm} E + T \\ &\Rightarrow_{rm} E + T * P \\ &\Rightarrow_{rm} E + T * a \\ &\Rightarrow_{rm} E + P * a \\ &\Rightarrow_{rm} E + a * a \\ &\Rightarrow_{rm} T + a * a \\ &\Rightarrow_{rm} P + a * a \\ &\Rightarrow_{rm} a + a * a \end{aligned}$$

□

7.2 Parser and parser-generator

A parser for a grammar G is a device which, given a string w in Σ^*, checks whether this string is in L(G) and, if so, outputs a possible derivation of w from S. Note that a string may have more than one derivation. □

A parser-generator is a device which, given a grammar G as input, produces a parser for G.

In practice a parser-generator will only accept a restricted class of grammars. A general system would be quite inefficient and not useable for practical purposes. At present the biggest and most natural class of grammars for which useable parser-generators/parsers can be constructed is the class of LR-grammars (in particular LALR(1)). We note that an ambiguous grammar is not an LR-grammar. □

7.3 LR(k) grammar

An LR-parser works by constructing a rightmost derivation backwards. The output of the parser is the reversed sequence of productions used in the rightmost derivation of the input string. This reversed sequence of productions is called a right parse. A sentential form is in the following always a rightmost sentential form.

If $S \Rightarrow^{*}_{rm} \alpha A w \Rightarrow \alpha \beta w \Rightarrow^{*}_{rm} x w$, where $x, w \in \Sigma^{*}$, then $\alpha \beta w$ is a rightmost sentential form which may appear during a parse of $x w$. β is called the handle of $\alpha \beta w$, and $A \rightarrow \beta$ is called the handle production. Given a sentential form, the purpose of the parser is to determine the handle production.

In general the whole sentential form must be available in order to determine the handle production. This is not desirable in practice as the input string may be very long. An LR(k) grammar has the property that the handle production of each sentential form can be uniquely determined by scanning the sentential form from Left to right, producing a Right parse, but only looking ahead at most k-symbols beyond the right end of the handle.

Let G be an LR(k) grammar and let $\alpha \beta w$ be a sentential form with handle production $A \rightarrow \beta$. If $\alpha \beta y$ is another sentential form, and if the first k symbols of w are identical with the first k symbols of y, then $A \rightarrow \beta$ must also be the handle production of $\alpha \beta y$.

This may be formalized in the following definitions:

Definition

Let $G = (\Sigma, N, P, S)$, and $\alpha \in (N \cup \Sigma)^{*}$, then

$\text{FIRST}_k(\alpha) = \{ x \in \Sigma^{*} \mid \alpha \Rightarrow^{*} xw \mid |x| = k$ or $\alpha \Rightarrow^{*} x, |x| < k \}$

$|x|$ is the length of the string x, $|e| = 0$ □

Definition

Let $G = (N, \Sigma, P, S)$ be a CFG. Let $G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S')$ be its augmented grammar. G is LR(k), $k \geq 0$, if the three conditions

1. $S' \Rightarrow^*_{rm} \alpha A w \Rightarrow \alpha \beta w$,
2. $S' \Rightarrow^*_{rm} \gamma B x \Rightarrow \gamma \delta x = \alpha \beta y$, and
3. $FIRST_k(w) = FIRST_k(y)$,

imply that $\alpha = \gamma$, $A = B$, and $x = y$ □

There are various technical reasons for using a new start symbol S', and a new production $S' \rightarrow S$. See Aho and Ullman [20].

7.3.1 Shift-reduce parser

In the following we shall only consider LR(1) grammars. An LR-parser belongs to a class of parsers which all work in a similar way. Below we describe these parsers.

The parser uses a stack called the parse stack.

It works as follows:
Initially the stack is empty and input consists of the string w:

Let w be divided into $x a y$ with x, y in Σ^* and a in Σ. At a given point during the parse the x part has been read and processed and the remainder of the input is ay. The parse stack will then contain a string α in $(N \cup \Sigma)^*$, such that

$$\alpha a y \Rightarrow^* w$$

and such that α a y is a sentential form in the right most derivation of w from S. Productions used in reducing x to α have been outputted. The situation is:

stack	input
α	a y

The parser determines its next step on the basis of α and a (i.e. not using information about y). There are four different cases:

- shift: read a and shift it onto the stack.

stack	input
α a	y

- reduce: perform a reduction, that is α a y has the form $\gamma\,\delta$ a y and $B \rightarrow \delta$ is a production and γ B a y $\Rightarrow_{rm} \gamma\,\delta$ a y is a step in the rightmost derivation of w from S. The production $B \rightarrow \delta$ is outputted, and the situation is:

stack	input
γ B	a y

- accept: the stack only contains the start symbol S, and input is empty.

- error: none of the above cases is applicable; w is not in L(G).

□

To summarize, the parser shifts input symbols onto the stack, until a handle appears on top of the stack. It then performs a reduction. If no errors occur, this proces is continued until the start symbol appears on the stack. Parsers working in this way are called shift-reduce parsers.

Example 7.3

Consider a parse of the string a + a * a in example 7.2.

Action	Stack	Input	Output
		a + a * a	
shift	a	+ a * a	
reduce	P	+ a * a	P → a
reduce	T	+ a * a	T → P
reduce	E	+ a * a	E → T
shift	E +	a * a	
shift	E + a	* a	
reduce	E + P	* a	P → a

Action	Stack	Input	Production
reduce			
	E + T	* a	T → P
shift			
	E + T *	a	
shift			
	E + T * a		
reduce			
	E + T *P		P → a
reduce			
	E + T		T → T * P
reduce			
	E		E → E + T
accept			

□

7.3.2 Construction of LR-parsers

In practice the LR-parser does not just 'look at' the stack and the next input symbol to decide what to do next. The parser keeps track of those productions which are possible candidates for a reduction during the parse. This is done using tables, which specify the next move at each state of the parse. These tables are constructed by the parser generator.

Let $A \to \beta_1 \beta_2$ be a production. Consider the derivation

$S \Rightarrow^*_{rm} \alpha A a y \Rightarrow \alpha \beta_1 \beta_2 a y \Rightarrow^*_{rm} \alpha \beta_1 x a y \Rightarrow^*_{rm} v x a y,$

where $a \in \Sigma$, $y \in \Sigma^*$ or $a = e$, $y = e$.

During a parse of $v x a y$, $\alpha \beta_1$ will appear on the stack at some stage with $x a y$ on input. When x has been reduced to β_2, then $A \to \beta_1 \beta_2$ is the handle production and a is the next input symbol. In general with $\alpha \beta_1$ on the stack we know nothing about the input. But if part of the input is reduced to β_2 and a is the next input symbol, then $A \to \beta_1 \beta_2$ is the handle production. This information that $A \to \beta_1 \beta_2$ is a future candidate for a reduction when the stack contains $\alpha \beta_1$ is remembered in the following way:

> $\alpha \beta_1$ is called a viable prefix and we say that $[A \to \beta_1 \, . \, \beta_2, a]$ is an LR(1) - item (or just LR-item) which is valid for $\alpha \beta_1$.

In general a viable prefix has a lot of valid LR(1) - items. The set of valid LR(1)-items for a viable prefix $\alpha \beta_1$ is

$$V_1 (\alpha \beta_1) = \{ [B \to \delta' \, . \, \delta'', b] \mid B \to \delta' \delta'' \text{ is a production, and } S \Rightarrow^*_{rm} \gamma B z \Rightarrow \gamma \delta' \delta'' z = \alpha \beta_1 \delta'' z, \text{ and } b \in FIRST_1 (z) \}$$

For a viable prefix $\alpha \beta_1$, $V_1(\alpha \beta_1)$ contains sufficient information to decide what parsing action to take if $\alpha \beta_1$ is on the stack.

Let $\alpha \beta_1$ be on the stack, let x be the next input symbol and let $[A \to \beta_1 . \beta_2, a]$ be in $V_1(\alpha \beta_1)$. There are five cases, dependent on β_2:

1. $\beta_2 = x\beta_2'$.

   I. e. the first symbol of β_2 is identical to the input symbol, and the parser has to do a shift. The next state of the parse is described by $V_1(\alpha\beta_1 x)$, and $[A \to \beta_1 x.\ \beta_2', a] \in V_1(\alpha\beta_1 x)$.

2. $\beta_2 = y\beta_2'$, $y \in \Sigma$, $y \neq x$.

   $A \to \beta_1\beta_2$ is no longer a candidate for a reduction.

3. $\beta_2 = B\beta_2'$, $B \in N$.

   $A \to \beta_1\beta_2$ is still a candidate for a reduction, but we must first recognize a production of the form $B \to \delta$. If and when part of the input has been reduced to B, then the next state of the parse is described by $V_1(\alpha\beta_1 B)$ and $[A \to \beta_1 B\ .\ \beta_2', a] \in V_1(\alpha\beta_1 B)$. It also follows that $\{ [B \to .\ \delta, b] \mid B \to \delta$ is a production, and $b \in FIRST_1(\beta_2' a) \} \subseteq V_1(\alpha\beta_1)$.

4. $\beta_2 = e$ (the empty string), $x = a$

   We may reduce by $A \to \beta_1$, and return to the state $V(\alpha)$, with α on the stack and shift A on the stack. The next state to consider is then $V(\alpha A)$.

5. $\beta_2 = e$, $x \neq a$.

   $A \to \beta_1$ is no longer a candidate for a reduction.

□

For an LR(1) - grammar there are no conflicts between different items of $V_1(\alpha\beta_1)$ in deciding whether to shift or reduce.

For a non LR(1) grammar there will exist a viable prefix φ such that $V_1(\varphi)$ contains two items of the form

$$[A \to \beta_1 .\ a\beta_2, b],\ [B \to \beta., a],\ a \in \Sigma,$$

or two different items of the form

$$[A \to \alpha ., a],\ [B \to \beta\ .\ , a],\ a \in \Sigma \cup \{e\}.$$

In both cases there is a conflict on the input symbol a. $V_1(\varphi)$ is said to be <u>inconsistent.</u>

□

For a given set of LR-items $V_1(\alpha)$ we define two functions – the parsing action function PA, and the successor-function SUCC.

Definition

Let α be a viable prefix of a sential form of an LR(1) grammar $G = (N, \Sigma, P, S)$. Let $x \in \Sigma \cup \{e\}$. The parsing action function

$PA(V_1(\alpha), x)$ takes one of the values

- shift, if an item $[A \rightarrow \beta_1 . x\beta_2, a] \in V_1(\alpha)$,
- reduce $A \rightarrow \beta$, if an item $[A \rightarrow \beta., x] \in V_1(\alpha)$,
- accept, if $x = e$ and $[S' \rightarrow S., e] \in V_1(\alpha)$, or
- error, if none of the above cases are applicable.

□

Definition

Let $T = V_1(\varphi)$ be a set of LR-items for an LR(1) grammar $G = (N, \Sigma, P, S)$. Let $x \in N \cup \Sigma$. $T' = V_1(\varphi x) = SUCC(T, x)$ is the set of items computed in the following way:

1. The basis of T' is computed by
   If $[A \rightarrow \alpha . x\beta, a]$ is in T,
   then $[A \rightarrow \alpha x . \beta, a]$ is in T'.

2. The closure of T' is computed by

   2.1 If $[A \rightarrow \alpha . B\beta, a]$ is in T',
   $B \rightarrow \delta$ is in P and $b \in FIRST_1(\beta a)$,
   then $[B \rightarrow . \delta, b]$ is in T'.

   2.2 Repeat step 2.1 until no more new items can be added to T'.

□

As there are only a finite number of LR-items for a given grammar G, it is possible to compute in advance all the sets of LR-items which may appear during a parse. This is in fact done by the parser generator.

The canonical collection of sets of LR-items may be computed by the following algorithm.

Algorithm

Let $G = (N, \Sigma, P, S)$ be a CFG. Let $G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \to S\}, S')$ be the augmented grammar. The canonical collection of sets of LR-items $\mathcal{C}$ is computed by

1. Compute the initial set A_0 of $\mathcal{C}$.

   1.1 $[S' \to . S, e]$ is in A_0

   1.2 Compute the closure of A_0

2. Compute the succeeding sets of $\mathcal{C}$

   2.1 Let A be in $\mathcal{C}$, and x in $N \cup \Sigma$, then compute SUCC(A, x) and add it to $\mathcal{C}$.

   2.2 Repeat step 2.1 until no new set can be added to $\mathcal{C}$.

□

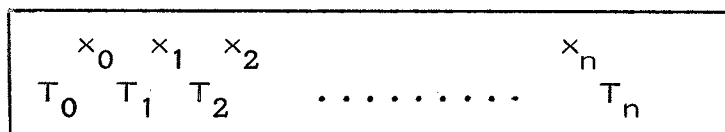
A set of items in the canonical collection of LR(1)-items will in the following be called an LR(1)-table (or just an LR-table). Note that this term is not that used by Aho & Ullman [20].

7.3.3 LR-parsing algorithm.

We may now formulate the parsing algorithm using LR-tables. Besides the symbols being stacked, the LR-tables will also be

stacked in order to find the next LR-table to be used after a reduction. If $PA(V_1(\alpha\beta), a)$ is reduce $A \rightarrow \beta$, then αA will be the new stack content and $V_1(\alpha A) = SUCC(V_1(\alpha), A)$.

In general the stack contains



where T_0 is the initial LR-table corresponding to the empty stack and

$$T_i = V_1(x_1 x_2 \ldots x_i),\ i = 1, 2, \ldots, n.$$

Algorithm LR(1) parsing algorithm.

```
BEGIN
        T : = initial LR-table;
        x : = next input symbol;
        let the stack initially contain T ;
  REPEAT
      CASE PA(T, x) OF
        shift   : BEGIN shift x onto the stack;
                        T : = SUCC(T, x) ;
                        shift T onto the stack;
                        x : = next input symbol
                  END;

        reduce A → β : BEGIN output A → β ;
                             pop 2 · |β| symbols off the stack;
                             T : = LR-table on top of the stack;
                             shift A onto the stack;
                             T : = SUCC (T, A)
                             shift T onto the stack;
                       END;
```

```
            accept :  announce accept;
            error  :  announce error
        END CASE;
    UNTIL accept OR error;
END;
```

□

7.4 Practical LR-grammars.

For large grammars it turns out that the size of the LR(1)-tables and the amount of work required to construct them is very large. However, if one considers a subset of the class of LR(1) grammars then there exist techniques for constructing usable LR-parsers. We shall consider two such methods.

In both methods one starts by constructing the sets of canonical LR(0)-items (LR(0)-tables) for the given grammar.

An LR(0) table T may be characterised in the following way, as a

- shift table, if T contains no item of the form $[A \to \beta\, .\, , e]$,
- reduce table, if T contains an item of the form $[A \to \beta\, .\, , e]$, and all other items have the form $[B \to \beta_1 .\, C\beta_2 , e]$.
- inadequate table, if T contains either two items of the form

  $[A \to \beta_1 .\, a\beta_2 , e]$, $[B \to \beta\, .\, , e]$

  or two different items of the form

  $[A \to \alpha\, .\, , e]$, $[B \to \beta\, .\, , e]$.

The parsing action of a shift table and of a reduce table can be determined independently of the next input symbol. This is not the case with an inadequate table where there is more than one parsing action which may be chosen. Inadequate tables correspond to inconsistent sets of items in the LR(1) case. Consequently a grammar is LR(0) if and only if none of its LR(0)-tables are inadequate.

In order to resolve these conflicts, each inadequate table is converted to a lookahead table. An attempt could then be made to try to resolve the conflicts by looking at the next input symbol.

Let $\alpha\beta$ be a viable prefix, let $T = V_0(\alpha\beta)$ be an inadequate LR(0)-table. Let the item $[A \to \beta.,e]$ be in T. A lookahead set $LA(T,[A \to \beta.,e]) \subseteq \Sigma \cup \{e\}$ is computed. The idea is that if, given a parse state described by T, it is possible to reduce using the rule $A \to \beta$, then the next input symbol will be in $LA(T,[A \to \beta.,e])$. This can be expressed by

$$LA(T,[A \to \beta.e]) \supseteq \{a \mid \alpha'\beta w \text{ is a sentential form with with } V_0(\alpha'\beta) = T,\ a \in FIRST_1(w)\}$$

Below we describe two types of LR-grammars which differ in the way LA is defined.

Having computed $LA(T,[A \to \beta.,e])$, the item $[A \to \beta.,e]$ is replaced by $\{[A \to \beta.,a] \mid a \in LA(T,[A \to \beta.,e]\}$. By doing this for all items in T which may lead to a reduce, it is possible to obtain a lookahead table T_L from T. The parsing conflicts to T are then resolved if the parsing action function PA can be uniquely defined on T_L.

There exist LR(1)-grammars for which the above way of constructing LR-tables does not work.

7.4.1 Simple LR(1) grammars.

We now consider one way of defining LA.

Definition

Let T be an LR(0)-table. The set of symbols for which the parsing action is shift is:

$$SHIFT(T) = \{a \in \Sigma \mid T \text{ contains an item of the form } [A \to \beta_1 . a\beta_2, e]\}$$

□

Definition

Let $G = (N, \Sigma, P, S)$ be a CFG. Let $A \in N$, then

$$FOLLOW_1(A) = \{ a \in \Sigma^* \mid s \Rightarrow^* \alpha A w, \ a \in FIRST_1(w) \}$$

□

Definition

Let $G = (N, \Sigma, P, S)$ be a CFG. G is said to be Simple LR(1) (SLR(1)) if any inadequate LR(0) table satisfies the following conditions

(1) If $[A \to \beta . , e]$ is in T, then

$$FOLLOW_1(A) \cap SHIFT(T) = \emptyset$$

(2) If $[A \to \alpha . , e]$ and $[B \to \beta . , e]$ is in T, then

$$FOLLOW_1(A) \cap FOLLOW_1(B) = \emptyset$$

□

If $[A \to \beta . , e]$ is an item in an inadequate table T, then $FOLLOW_1(A)$ may be used as $LA(T, [A \to \beta . , e])$.

We can define SLR(1) grammars independently of the LR(0) tables. Let G be an SLR(1) grammar, let $\alpha \beta w$ be a sentential form with reduction handle $A \to \beta$. If $\alpha \beta x$ is a sentential form and $FIRST_1(x)$ is in $FOLLOW_1(A)$, then $A \to \beta$ must be the reduction handle for $\alpha \beta x$.

The lookahead sets of an SLR(1) grammar are not minimal. This is because FOLLOW is computed independently of the given inadequate state.

7.4.2 Lookahead LR-grammars.

We shall now consider the case with minimal lookahead sets.

If T is an LR(1)-table, then we define

$$\mathrm{CORE}(T) = \{ [A \to \alpha . \beta, e] \mid [A \to \alpha . \beta, a] \in T \text{ for some } a \}$$

If T_0 is an LR(0)-table for a grammar G, then there exists an LR(1) table T for G such that $T_0 = \mathrm{CORE}(T)$, and vice versa.

<u>Definition</u>

Let $G = (N, \Sigma, P, S)$ be a CFG. Let $\mathcal{C}_0$ be the set of LR(0)-tables and let $\mathcal{C}_1$ be the set of LR(1)-tables. Let T be in $\mathcal{C}_0$, and let T contain an item of the form $[A \to \beta . , e]$. We define

$$\mathrm{LALR}_1(T, [A \to \beta . , e]) = \{ a \in \Sigma \mid [A \to \beta . , a] \in T', \ T' \in \mathcal{C}_1, \text{ and } \mathrm{CORE}(T') = T \}$$

□

<u>Definition</u>

Let $G = (N, \Sigma, P, S)$ be a CFG. G is said to be <u>Lookahead LR(1)</u> (LALR(1)) if any inadequate table T satisfies the following conditions:

(1) If $[A \to \beta . , e] \in T$, then
$\mathrm{LALR}_1(T, [A \to \beta . , e]) \cap \mathrm{SHIFT}(T) = \emptyset$

(2) If $[A \to \alpha . , e]$ and $[B \to \beta . , e] \in T$, then
$\mathrm{LALR}_1(T, [A \to \alpha . , e]) \cap \mathrm{LALR}_1(T, [B \to \beta . , e]) = \emptyset$

□

The lookahead sets $\mathrm{LALR}_1(T, [A \to \beta . , e])$ can be computed directly from the LR(0)-tables without computing the LR(1)-tables. An algorithm for doing this is given in [23].

It may be difficult for an unexperienced reader to distinguish between the various types of grammars.

In practice the following rules seem to apply:

- It often happens that an LR(1) grammar which is not SLR(1) is LALR(1).
- It is not very common that a grammar which is not LALR(1) is in fact LR(1). Usually such a grammar is ambiguous.

We end this section with an example of an LALR(1) grammar which is not SLR(1).

<u>Example</u>

Consider the grammar

$G = (\{S', S, A\} , \{ a, b, c, d, f \}, P, S')$ where P consists of

$S' \rightarrow S$
$S \rightarrow aAd$
$S \rightarrow afc$
$S \rightarrow bAc$
$S \rightarrow bfd$
$A \rightarrow f$

The LR(0) tables for G are

($\rightarrow$ indicates the SUCC function)

State	Items	Transitions
T_0	$S' \to .S$ $S \to .aAd$ $S \to .afc$ $S \to .bAc$ $S \to .bfd$	$\stackrel{S}{\to} T_1$ $\stackrel{a}{\to} T_2$ $\stackrel{b}{\to} T_7$
T_1	$S' \to S.$	
T_2	$S \to a.Ad$ $S \to a.fc$ $A \to .f$	$\stackrel{A}{\to} T_3$ $\stackrel{f}{\to} T_5$
T_3	$S \to aA.d$	$\stackrel{d}{\to} T_4$
T_4	$S \to aAd.$	
T_5	$S \to af.c$ $A \to f.$	$\stackrel{c}{\to} T_6$
T_6	$S \to afc.$	
T_7	$S \to b.Ac$ $S \to b.fd$ $A \to .f$	$\stackrel{A}{\to} T_8$ $\stackrel{f}{\to} T_{10}$
T_8	$S \to bA.c$	$\stackrel{c}{\to} T_9$
T_9	$S \to bAc.$	
T_{10}	$S \to bf.d$ $A \to f.$	$\stackrel{d}{\to} T_{11}$
T_{11}	$S \to bfd.$	

G is not LR(0) since T_5 and T_{10} are inadequate.

G is not SLR(1) since $FOLLOW_1(A) = \{c, d\}$.

G is LALR(1) since $LALR_1(T_5, [A \to f., e]) = \{d\}$
and $LALR_1(T_{10}, [A \to f., e]) = \{c\}$

□

REFERENCES

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

[2] DeRemer, F.L.
"Simple LR(k) grammars"
Comm. ACM 14, 453 - 460 (1971)

[3] Knuth, D.E.
"On the translation of languages from left to right"
Information and Control 8, 607 - 639 (1965)

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

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

[6] Wirth, N.
"The programming Language Pascal"
Acta Informatica, 1, 35 - 63 (1971)

[7] Jensen, B.B., Madsen, O.L., Kristensen, B.B. and Eriksen, S.H.
"BOBS-System Brugervejledning"
DAIMI PB-10 (in Danish), March 1973
DAIMI, University of Aarhus

[8] Kristensen, B.B., Madsen, O.L., Jensen, B.B. and Eriksen, S.H.
"A Short Description of a Translator Writing System (BOBS-System)"
DAIMI PB-11, February 1973
DAIMI, University of Aarhus

[9] Kristensen, B.B., Madsen, O.L., Jensen, B.B. and Eriksen, S.H.
"BOBS-System. Tilføjelser og ændringer til brugervejledning"
DAIMI PB-22 (in Danish), January 1974
DAIMI, University of Aarhus

[10] Kristensen, B.B., Madsen, O.L., Jensen, B.B. and Eriksen, S.H.
"A Short Description of a Translator Writing System (BOBS-System)"
DAIMI PB-41, October 1974
DAIMI, University of Aarhus

[11] Burger, W.F.
"BOBSW - A Parser Generator"
SESLTR-7, December 1974
Department of Computer Science
The University of Texas At Austin

[12] Jensen, K. and Wirth, N.
"Pascal User Manual and Report"
Springer Verlag, 1975

[13] Kristensen, B.B., Madsen, O.L. and Jensen, B.B.
"An Implementation of a Pascal Compiler"
Unpublished paper, April 1974
DAIMI, University of Aarhus

[14] Kristensen, B.B.
"Erkendelse og korrektion af syntax fejl under LR-parsing"
Masters Thesis (in Danish), May 1974
DAIMI, University of Aarhus

[15] Jensen, B.B.
"An Extension of the BOBS-System to a full Compiler-Compiler Based on Mathematical Semantics"
Masters Thesis, February 1975
DAIMI, University of Aarhus

[16] Mosses, P.
"The Mathematical Semantics and Compiler Generation"
Ph.D. Thesis, September 1974
Oxford University Programming Research Group
Oxford, England

[17] Madsen, O.L.
"On the Use of Attribute Grammars in a Practical Translator Writing System"
Masters Thesis, July 1975
DAIMI, University of Aarhus

[18] Aho, A.V. and Johnson, S.C.
"LR-Parsing"
Computing surveys 6, 99 - 124 (June 1974)

[19] Eve, J.
"On the Recovery of LR(1) Parsers from Computed Parsing Tables"
Computing Laboratory
The University of Newcastle upon Tyne
England

[20] Aho, A.V. and Ullman, J.D.
"The Theory of Parsing, Translation and Compiling"
Volume 1: Parsing, 1972
Volume 2: Compiling, 1973
Prentice Hall, Englewood Cliffs (N.J.)

[21] Knuth, D.E.
"Semantics of Context Free Languages"
Mathematical Systems Theory, vol. 2 (1968) 127 - 145

[22] Lewis, P.M., Rosenkrantz, D.J. and Stearns, R.E.
"Attributed Translation"
Journal of Computer and System Sciences 9, 279 - 307 (1974)

[23] Kristensen, B.B., Madsen, O.L.
"Methods for Computing LALR(k) Lookahead"
ACM Transactions on Programming Languages 3, 1, 60-82 (1981).

Appendix A

This appendix describes how to use the BOBS-system on the CDC 6400 at the Regional EDP-Center (RECAU), and on the DEC-10 at the Department of Computer Science (DAIMI) both at the University of Aarhus.

We shall describe the job control cards used to call the system and the means by which the PASCAL files INPUT, OUTPUT, PARSIN, PARSOUT and TABLES are interfaced to the file systems of the respective machines. The parser generator has the following program head:

```
PROGRAM BOBS (PARSIN, PARSOUT, TABLES, INPUT,
                                        OUTPUT);
```

The skeleton compiler has the following program head:

```
PROGRAM BOBS (INPUT, OUTPUT, TABLES ) ;
```

A1 The BOBS-system at RECAU

At the time of writing RECAU is using the KRONOS operating system on a CDC 6400. In the near future it is planned to change to the operating system NOS, and to change to a CDC CYBER 173. The BOBS system should, however, not be affected.

The direct permanent file BOBSSYS/UN = DATBOBS contains the parser generator. BOBSSYS is a loadable file generated by the PASCAL compiler. The indirect permanent file PARSIN/UN = DATBOBS contains the skeleton compiler.

The file names of both the parser generator and skeleton compiler may be substituted at execute time according to the usual KRONOS rules.

The following is a KRONOS job concomitant to the examples 4. 1 and 5. 1. The file DEMGRAM contains the input to the parser generator. The file DEMSTRG contains the input to the skeleton compiler.

```
DATZZ,CM50000,T100.                *BOBSSYSTEM DEMO EXAMPLE
USER,DATBOBS,          .           *
CHARGE,AAUDATRF,BOBS.              *
ATTACH,BOBSSYS/UN=DATBOBS.         *PARSER-GENERATOR
GET,PARSIN/UN=DATBOBS.             *SKELETON COMPILER
GET,DEMGRAM.                       *INPUT GRAMMAR
REDUCE,-.                          *
BOBSSYS,,,,DEMGRAM.                *RUN PARSER-GENERATOR
REWIND,PARSOUT.                    *
PASCAL,I=PARSOUT.                  *COMPILE SKELETON COMPILER
GET,DEMSTRG.                       *STRING TO BE PARSED
LGO,DEMSTRG.                       *RUN SKELETON COMPILER
SAVE,PARSOUT.                      *SAVE SKELETON COMPILER
SAVE,TABLES.                       *SAVE PARSE TABLES
```

A2 The BOBS-system at DAIMI

DAIMI is currently using the TOPS-10 operating system on a PDP-10.

The parser generator is called by the command R BOBS.

You will then be asked to specify the following files:

PARSIN	=	(the skeleton compiler, normally SYS : PARSIN)
PARSOUT	=	(the generated skeleton compiler)
TABLES	=	(the generated parse tables)
INPUT	=	(the input grammar)
OUTPUT	=	(the output listing from the parser generator).

If you don't specify some of the file names (i. e. just type a NEWLINE), then you get default names in accordance with the PASCAL system on the PDP-10.

PARSOUT may be compiled and executed by the PASCAL system as a normal PASCAL program. The file TABLES in the program head must be specified at execute time to be the one containing the parse tables.

Appendix B, A (minor) extension to the BOBS-system

When adding semantics to the BOBS skeleton-compiler it has always been inconvenient that the user has no control over the numbering of the productions.

The parser generator is now able to generate a semantic interface which is a sequence of text lines, one for each production. Each line has the form:

PRODNO ␣ INPUTNO ␣ LABEL

PRODNO	is the number of the production as generated by the BOBS-system,
INPUTNO	is the number of the production obtained through a consecutive numbering of the productions in the order as they appear on input,
LABEL	is a string of characters which the user may attach to the production. See below for the way of doing this.

The semantic interface is written as the last information on file TABLES. When the procedure INITIALIZE has been called in the Skeleton Compiler, the file TABLES is positioned at the beginning of the semantic interface (some end-of-lines and blanks appear first).

If INPUTNO is wanted, option 33 must be used.

If LABEL is wanted, option 34 must be used, and an <alternative> in a <grammar-rule> (cf. 4.2.4) must be

<alternative> → <label> {<nonterminal> | <terminal>}$^+$

<label> → sequence of characters M1

The "sequence of characters" must not contain the current M1. Blanks are not skipped. No more than 100 chars may appear.

INPUTNO and LABEL will appear on the grammar listing.

Example

```
*********** A LIST OF INPUT WITH POSSIBLE ERRORMESSAGES ************

 OPTIONS(33,34)
 METASYMBOLS M1== M2=/ M3=< M4=;
 NAME KONST ( ) + * ;
 <E< = PLUS= <E< + <T< / = <T< ;
 <P< = ID=NAME / C=  KONST/=( <E< ) ;
 <T< = MULT= <T< * <P< / = <P< ;;

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

****************************** THE GRAMMAR ******************************


 1   <E> ::= <E> + <T>                                   :1 PLUS
 2           / <T>                                       :2

 3   <T> ::= <T> * <P>                                   :6 MULT
 4           / <P>                                       :7

 5   <P> ::= NAME                                        :3 ID
 6           / KONST                                     :4 C
 7           / ( <E> )                                   :5
*************************************************************************
```

The semantic interface as written on file tables:

```
1 1 PLUS
2 2
3 6 MULT
4 7
5 3 ID
6 4 C
7 5
```