XBSW 3.3

A PARSER GENERATOR

Version November 1981

TR 78-87

by Wilhelm F. Burger

Software Systems International, Inc.
2500 Q Street NW 310
Washington, DC 20007
(202) 337-4335

Copyright (C) 1979,1981

Table of Contents

1. Introduction 1

2. The XBSW System 3
   2.1 Input Module 3
   2.2 Table Generation Module 4
   2.3 Optimization Module 5
   2.4 Transformation Module 6
   2.5 Output Module 6

3. Input to the XBSW System 7
   3.1 Terminal Section 7
   3.2 Production Section 11

4. The Parser 14
   4.1 Parse Tables and Parsing Algorithm 14
   4.2 Error Recovery 16
   4.3 Error Recovery by Error Productions 17

5. The Lexical Scanner 18

6. The Skeleton Compiler 20
   6.1 Parser Interface 20
   6.2 Semantic Interface 21
   6.3 Interactive Usage 23

7. Some Results 24

References 25

Appendix A
   Options A-1

Appendix B
   Constants of the Parse Table Generator B-1

Appendix C
   Parse Table Generation (Example) C-1

Appendix D
   XBSW System on DEC-10 and DEC-20 . . . . D-1

Appendix E
   XBSW System on CDC Cyber E-1

Appendix F
   XBSW System on IBM-370 . . . . F-1

Appendix G
   XBSW System on VAX/VMS G-1

1. Introduction

The XBSW system is a general tool for quickly implementing the front-end of applications which require language-type input. The system can be used to advantage for the implementation of

- compilers,
- data management languages,
- special purpose languages.

There are numerous advantages in using an automated system over hand-coding. A few of these advantages are:

- reduced implementation time and cost,
- reduced probability of error,
- high degree of machine independence,
- easy modifiability of the parser when the language specification changes,
- compact code due to the use of tables.

The parsing speed of the generated parser is comparable to the speed of a hand-coded recursive descent parser.

The XBSW system is an enhanced version of the BOBSW 3.0 system [7]. This system is based on the ideas of DeRemer [1], LaLonde [2], and Aho and Ullman [3,4,5]. It is a parser generator for LR(0), SLR(1), and LALR(1) grammars. These grammars have the property that by looking at most one symbol ahead in the input string, it can be decided whether a reduction can be made or more input is required. This has the obvious advantage that the input string can be parsed without backtracking. Ambiguous grammars are also accepted by the XBSW system if disambiguation rules are given which assure the one symbol lookahead property.

The system consists of three major parts: the table generator, table transformation programs, and skeleton compilers. The table generator produces for a given grammar a machine and language independent parse table. This table is then transformed by a table transformation program to suit a particular skeleton compiler. Lexical scanner information is also generated in this step. The skeleton compiler is available in several languages and on different machines. The skeleton compiler consists of a lexical scanner and a parser with error recovery.

The XBSW 3.3 system is at present available on CDC Cyber, DEC-10, DEC-20, Burroughs 6700, Prime, IBM 370, and VAX machines. Skeleton compilers are available also for other machines, e.g. the 8080 micro-processor. The XBSW system has been and is being used in industrial and academic environments for a variety of projects. Compilers for Gypsy [13], Alphard [14], and the DoD-1 language ADA [15,16] are based on this system. The system has been used for query languages, e.g. FAST [17], data base languages, e.g. E/S-DBMS [18] and APRIL[19], and for special purpose languages, e.g. GAMS [20].

Section 2 describes the XBSW system briefly. Section 3 gives detailed instructions on how to use the system. The table structures and the parsing algorithm are described in section 4. Lexical scanner and skeleton compilers are handled in section 5 and 6. Some results on table sizes for various grammars are included. Special attention is given to the generation of tables which can be used with micro-computers [9,10]. The appendix contains an example for a subset of Pascal suggested in [5].

The use of the XBSW system is best learned by examples. Several grammars and skeleton compilers are delivered with the system for this purpose. The grammars which are usually provided include grammars for ADA, Pascal, and C. Some of the important aspects which are demonstrated by the examples are:

- the formulation of grammars,
- the use of semantic actions for syntax directed translation,
- the interactive use of the parser in conjunction with a command language.

The examples are provided as self-contained control card files (or control command procedures) where feasible. Local documentation should be consulted for their use.

2. The XBSW System

The table generator of the XBSW system consists of five modules. The modules perform the following steps:

I. Input
- input of grammar
- grammar checks
- grammar modifications

II. Table Generation
- generation of LR(0) machine
- generation of lookahead transitions for SLR(1) or LALR(1) grammars and removal of ambiguities
- generation of lookback states

III. Optimization
- simple production removal
- table compaction
- generation of error numbers

IV. Transformation
- restructuring of tables

V. Output
- output of parser tables

2.1 Input Module

Input consists of two parts: Definition of terminal symbols and specification of grammar productions. The proper input syntax is described in Section 3. Terminal symbols are grouped into tokens and lexemes. Tokens are strings of characters which represent themselves, e.g. reserved words of the language. Lexemes stand for a class of strings which are recognized by some lexical scanner algorithm, e.g. identifiers and numbers. Lexemes may be introduced also for other purposes, e.g. to handle some context-sensitive aspects of a language. Tokens are used to generate tables for the lexical scanner.

The grammar is formulated in a notation similar to BNF. Each production may be associated with a semantic action number. Additional information may be required for a production, e.g. a set of disambiguation symbols.

Other input facilities include the definition of precedence levels of terminal symbols and the definition of alternate goal symbols of the grammar.

After the input is read several consistency checks are carried out. All nonterminals (except the goal symbols) must appear in both the left side and the right side of some production. Each nonterminal must be able to

produce a string consisting of only terminal symbols. A non-fatal warning is given if not all terminal symbols are used in the grammar. Nonterminals can be both left and right recursive if the grammar is processed as ambiguous grammar. Nonterminals which are not reachable from the goal symbols are detected when the LR(0) machine is constructed.

Some grammar modifications may also take place. Identical productions are removed. Simple productions of the form <A> ::= <B> are eliminated if <A> does not appear on the left side of any other production, and if no semantic action number is associated with the production. If requested the grammar can be modified so that no nonterminal can produce the empty symbol.

2.2 Table Generation Module

The LR(0) machine is generated according to DeRemer and described in [1]. (Other references are Aho and Ullman [5] and Thompson [8]). In essence, it is a finite state machine which scans from left to right until it finds the leftmost reduction. It makes the reduction and then continues scanning for the next leftmost reduction. A pushdown stack is used to eliminate unnecessary rescanning after each reduction.

A grammar is LR(0) if no shift/reduce or reduce/reduce conflicts occur. States with conflicts are called "inadequate" states. Conflicts are created by states which must make a choice between a transition under a terminal symbol and a reduction, or between several reductions.

In many cases it is possible to resolve the conflict by looking at the terminal following the nonterminal of the reduction. In essence an attempt is made to build disjoint sets of terminals which direct the state transition. There are two ways to obtain one-symbol lookahead sets. The terminals following a nonterminal are collected according to the grammar. The grammar is SLR(1) if this set is sufficient to solve the conflict. A more restricted lookahead set can be found by considering also the state of the machine. The grammar is LALR(1) if conflicts can be resolved in this manner. Additional disambiguation rules can be defined to partition the lookahead sets (obtained by either method) to resolve the conflicts [3].

Disambiguation rules can be defined using precedence levels. Terminals are associated with an internal precedence number in the precedence section. Each production is given a precedence number according to the last terminal in the right side. (Precedence levels may be also explicitly assigned to productions). A read transition is carried out if the precedence of the terminal is higher than the precedence of the production calling for a reduction. If the precedence levels are the same then additional properties are required, namely if the terminal is right associative, left associative, or if it is a binary terminal. In the first case a read transition, and in the second case a reduce transition is carried out. If the terminal is binary then an error state is entered.

Another way to supply disambiguation information is by defining the set of terminal symbols which are to be used for read transitions, reduce transitions, or error transitions. The disambiguation symbols are defined together with the production causing the conflict.

If it is not possible to resolve the conflicts with one of the above methods then the grammar does not have the one-symbol lookahead property. The program stops if states with conflicts exist which could not be resolved. The grammar must be rewritten to achieve the one-symbol lookahead property.

Transitions under nonterminals are eliminated by introducing lookback states. (This corresponds to "goto" tables in [5]). Whenever a read state is entered, its state number is pushed onto the stack. After a reduction the state number on top of the stack is used to establish the context in which the reduction took place and which state is to be entered next. All this information can be precomputed. For each terminal a state is created which consists of the pairs: read state and next state to go to. Lookback states are only entered from reduce states.

2.3 Optimization Module

Simple productions which are not associated with a semantic action number are candidates for simple production removal. Only simple productions which are marked (see section 3.2.1) participate in the simple production removal process. The basic idea is described in [4]. Information associated with the simple production is distributed to other states so that a reduction with the simple production can be avoided. This has the effect that long chains of simple reductions are replaced by just one transition.

Space optimizations are realized by introducing default transitions. For example a lookahead state might contain a transition to a given state for several symbols. The largest set of those transitions is found and replaced by an unconditional transition to this state. This does not influence the error detection ability of the state. In a few cases, however, due to resolving ambiguities with error transitions, this optimization may not be possible.

The same optimization can be carried out for lookback states. Here the largest set of transitions to the same state is replaced by an unconditional transition to this state. If the lookback state contains only one entry then the state is deleted and the information distributed to the corresponding reduce states.

Another space optimization can be achieved by overlapping read transitions. A simple algorithm is used which overlaps the tail ends of read states if this is possible. The transformation module provides a more sophisticated algorithm which can achieve additional space savings.

Each read state has associated with it a set of expected terminal symbols. These sets are identified and numbered. The number is used as

error message when the input symbol in the respective state is not in the set of expected symbols.

2.4 Transformation Module

The states generated so far are represented by linked lists. This structure is transformed into a more efficient representation. A state table is created which uses positional information to encode the state type, e.g. if it is a read, lookback, or reduce state. Information which is associated with read and looback states is stored in a separate table. The table structures are described in section 4.

When placing the read states into the new table structure additional space optimizations can be carried out. An algorithm is implemented which finds a near optimal overlap of read transitions. This optimization is optional.

The lookback states are always transformed into a "direct lookup" structure. A heuristic was developed to interleave lookback states so that they require the least amount of space. The structure of lookback states is described in section 4.

2.5 Output Module

The generated parser tables are written to a file in a language and machine independent form. The file is written in two parts. First the lexical scanner information is provided. It consists of the terminal symbols together with their internal encoding which is used by the parser tables. This is followed by the parser tables, which are a collection of numbers. Fortran-style formatting information precedes each sub-section to help with further processing.

Other output is generated for the user's information. Most of the output is controlled by options. The following lists only the most important output. The grammar is "pretty"-printed in Backus-Naur form (together with semantic action numbers if they were provided). Error numbers together with their expected symbols are listed. Inadequate states are displayed. The space requirements of the system are reported. Finally information about the size of the generated tables and the internal encoding of lexemes is given.

3. Using the XBSW System

This chapter describes the input to the table generator. Each input section starts with a keyword. All keywords must begin in column 1. The input is format free except for column 1 which is used for special purposes. The sections must be in the following order:

OPTIONS	1
VERSION	2
EMPTY	3
CLASSES	4
TERMINALS	5
LEXEMES	6
SCANNER	7
SPECIAL	8
PRECEDENCE	9
STRINGCH	10
ENDCH	11
PRODUCTIONS	12

Any of the sections except TERMINALS and PRODUCTIONS may be omitted. Sections 1 to 11 are described in the "terminal section", section 12 is described in the "production section".

3.1 Terminal Section

The character set is divided into three groups:

1 letters and digits
2 all other characters except space
3 space

Terminals by default either consist of characters of group 1 or characters of group 2. A terminal which belongs to group 1 must start with a letter. Terminals may be up to 20 characters long. (The length can be changed by recompiling the table generator). Terminal symbols may consist of characters of group 1 and of group 2 if option 5 is used.

The character set accepted by the table generator depends on the machine where the generator is used. If necessary "place holder" terminals must be used for terminals which consist of characters which are not in the accepted character set. The place holder terminals appear in the generated table file. For further processing they must be replaced here with the actual terminals.

Comment lines may appear in the XBSW input. Comment lines in the terminal section start with an asterisk in column 1. Comments in the production section have a different form; they are described in 3.2.

OPTIONS

The keyword OPTIONS is followed by one or several integers separated by commas. The available options are listed in Appendix A.

Example: OPTIONS 1,2,26

VERSION

The text following on the same line as VERSION is passed to the skeleton compilers. The text is limited to 60 characters. It may be used to identify the grammar and its version.

Example: VERSION ADA-1 MARCH 79

EMPTY

The name for the empty symbol is defined in this section.

Example: EMPTY LAMBDA

CLASSES

Terminals (tokens as defined in 2.1) with the same syntactic properties can be grouped together into a syntax class. Only the first element of a class may be used to formulate productions. The beginning of a class is indicated by + in column 1.

```
Example:  CLASSES
          +  *  /  DIV  MOD
          +  +  -
```

Terminals are associated with an internal number which is used by the parser. All terminals in a class have the same number. Each terminal in a class is also associated with a "variant" number. Variant numbers are assigned to terminals in the sequence they appear in the input, starting with zero. The lexical scanner returns these two numbers when a terminal is recognized. Parse tables are smaller if terminals are grouped into classes.

TERMINALS

All terminals of the language (tokens as defined in 2.1) are defined in this section (unless they are already defined in a CLASS section).

```
Example:    TERMINALS
               BEGIN  END  IF  THEN  ELSE
               :=  ,  ;
```

Each terminal is associated with an internal number which is used by the parser. The variant number of a terminal in this section is 0.

LEXEMES

Lexemes are terminals for parsing purposes only. They are in general recognized with some lexical scanner algorithm. Lexemes can be used for a variety of purposes, e.g. to express certain context-sensitive aspects of a language.

The standard skeleton compilers are written with the assumption that three lexemes are defined which represent identifiers, numbers and strings.

```
Example:    LEXEMES
               IDENTIFIER  NUMBER  STRING
```

A particular error recovery algorithm is provided which uses explicit "error" productions. A fourth lexeme ERROR must be defined if this kind of error recovery is used (see section 4.3).

The standard skeleton compiler must be modified if more lexemes are used. For example, two lexemes IDENT and PIDENT may be defined if for a particular language the syntax of predefined procedures is different from user defined procedures. In this case the lexical scanner must consult the symbol table and decide if a name is to be given as IDENT or PIDENT to the parser.

SCANNER

Terminal symbols which must be handled by the lexical scanner but which do not appear in the grammar are defined in this section. The standard skeleton compilers assume that the symbol which starts a comment is defined here.

```
Example:    SCANNER
               (*
```

SPECIAL

Any of the terminals defined in the previous sections may be flagged to be special. This can be used to invoke immediate actions whenever the lexical scanner has recognized the terminal.

```
Example:   SPECIAL
             BEGIN  END
```

PRECEDENCE

This section defines the precedence of terminal symbols and their associativity. The precedence between terminal and production is used to disambiguate read/reduce conflicts. The precedence of a production is the precedence of its rightmost terminal. The precedence of a production may be redefined (see section 3.2.3).

A terminal is either right associative, left associative or it is binary. Terminals with the lowest precedence come first. A precedence definition starts out with a keyword in column 1. The keyword is either RIGHT, LEFT or BINARY. Terminals of the same precedence level are formulated together. If they require more than one line then any extra lines must start with a blank in column 1.

```
Example:   PRECEDENCE
           LEFT        OR
           LEFT        AND
           BINARY      =  <>
           LEFT        +  -
           LEFT        *  /
           RIGHT       **
```

STRINGCH

This section defines the string escape character. It is only useful if a lexeme of kind string is defined. The string escape character must be on the same line as the keyword. The standard skeleton compilers recognize a character sequence enclosed in the string escape character as string. The string escape character may be part of the string if it is immediately followed by another string escape character.

```
Example:   STRINGCH  "
```

ENDCH

The closing symbol of a comment is provided in this section. The symbol may consist of one or two characters. The symbol must be defined on the same line as the keyword. The terminal symbol which starts a comment is provided in the section SCANNER. If the section ENDCH is omitted then a comment (if defined) is delimited by the end of line.

```
Example:   ENDCH   *)
```

3.2 Production Section

Four metasymbols are required to formulate productions. Any character except blank may serve as metasymbol. The metasymbols must be different from each other. A metasymbol definition looks as follows:

```
METASYMBOLS M1='  M2=$  M3=/  M4=!
```

METASYMBOLS is a keyword which starts in column 1. The meta- symbols by default are: M1=', M2=$, M3=/, M4=!. The metasymbols are used as follows:

- M1 corresponds to the Backus-Naur notation ::= .
- M2 separates several right sides for the same nonterminal.
- M3 delimits nonterminals.
- M4 indicates the end of a sequence of right sides.

A fifth metasymbol is available when option 29 is used. This metasymbol defines the beginning of a comment in the production section. Comments are terminated by the end of line. The default symbol is M5=%.

Metasymbols may be redefined by a metasymbol definition anywhere in the grammar specification in order to avoid conflicts with terminals or nonterminals.

Nonterminals may consist of up to 30 characters and include blanks. A nonterminal may not contain the metasymbol M3 which is in effect at that time.

The goal symbol of the grammar may be defined before the productions are specified. The keyword GOALSYMBOL starts in column 1 and is followed by the respective nonterminal. If no goal symbol is defined then the nonterminal of the left side of the first production is used as goal symbol.

Additional goal symbols are defined in an ENTRIES section. The keyword ENTRIES starts in column 1. It is followed by the respective nonterminals. The list of nonterminals is ended by metasymbol M4. The additional goal symbols make it possible to enter the parser in different contexts, e.g. just to parse a statement in an interactive environment. Several grammars may be also combined this way in one table. See section 6.1 on how to enter a parser with several goal symbols. A production

```
<PG-GOAL> ::=  <GOALSYMBOL>  END-OF-FILE
```

is added to the grammar for each goal symbol. (No semantic action is associated with these productions, i.e. they have semantic action number 0).

```
Example:   PRODUCTIONS
           METASYMBOLS M1==  M2=$  M3=/  M4=!  M5=%

           GOALSYMBOL  /PROGRAM/
           ENTRIES  /STATEMENT/  /EXPR/  !
```

GOALSYMBOL and ENTRIES (in that order) must be specified before any productions are defined.

The right side of a production may consist of at most 7 terminal and nonterminal symbols. If an empty symbol is used then it must be the only symbol on the right side. Each production can be associated with a semantic action number. The number must immediately follow metasymbol M2 or M4. This number may not exceed CONST1 (see Appendix B). User supplied semantic action numbers are in effect if option 26 is used. The connection between semantic action numbers and semantic routines is described in section 6.2.

```
Example:     /EXPR/ = /EXPR/ + /TERM/ $10
                      /TERM/ !
             /TERM/ = /TERM/ * /FACTOR/ $11
                      /FACTOR/ !
             /FACTOR/ = IDENTIFIER $12
                        ( /EXPR/ )  !13
             /STMT LIST/ = LAMBDA $    % EMPTY SYMBOL
                           /STMT LIST/ /STATEMENT/ !
```

Note: The production section must be ended with metasymbol M4 in column 1.

3.2.1 Simple Production Removal

Simple productions which participate in the simple production removal process are marked with S in column 1 immediately following the production.

```
Example:      /TERM/ = /FACTOR/  $
           S
                       /TERM/ * /FACTOR/ !
```

(These simple productions must not be associated with a semantic action number).

3.2.2 Disambiguation Sets

A production may be followed by disambiguation sets. Disambiguation sets are used to resolve read-reduce conflicts. For a given symbol a choice can be made whether to continue reading, to perform a reduction, or to report an error. Column 1 is used to indicate the choice:

```
C   continue reading
R   reduce
E   report error
```

The symbols for which the choice is to be made follow on the same line. If more than one line is necessary then the letter of column 1 must be repeated. The choices must be in the above order.

```
Example:       /EXPR/  =  /EXPR/  +  /EXPR/  !
             C  *
             R  +
                /EXPR/  =  /EXPR/  *  /EXPR/  !
             R  *  +
```

Before disambiguation is used one should check carefully if an alternate formulation of the grammar is as desirable but which avoids the need for disambiguation sets.

3.2.3 Disambiguation by Precedence

Productions may be associated with a precedence. By default the precedence of a production is the precedence of its rightmost terminal (if the terminal is specified in the PRECEDENCE section). A precedence can be explicitly assigned to a production by associating it with a terminal of the desired precedence. The production is followed by P in column 1 and the terminal.

```
Example:       /EXPR/ = /EXPR/ + /EXPR/ $
             P  *
                ...
```

Any production can be assigned a precedence in this manner.

Disambiguation by precedence is a global mechanism wheras disambiguation by disambiguation sets is a local mechanism. The local mechanism is useful if no precedence is associated with a terminal, or if an artificial precedence would be required to make the global mechanism work. In resolving a conflict first local and then global disambiguation is applied. Disambiguation is a very powerful mechanism, and one has to check carefully that the effects are the desired ones.

4. Parser

4.1 Parse Tables and Parsing Algorithm

The parse tables consist of a state table and a check table. The state table is divided into three parts: read states, lookback states, and reduce states. The check table consists of two parts: the first part is associated with read states and the second part is associated with lookback states.

The parser actions together with the table structures for each state are described in the following. The parser initially starts in a read state.

Read State

In a read state the input symbol is compared with the expected symbols in this state. (The expected symbols of a state are stored in the check table). If the input symbol is found among the expected symbols then an action associated with the symbol is carried out. If the symbol is a "read" symbol then the input is shifted, the current state of the parser is pushed onto the stack, and the parser goes to the next state which is associated with the symbol. If the symbol is a "lookahead" symbol then the parser simply goes to the next state associated with the symbol. Each expected symbol thus carries two pieces of information: the property of being a lookahead or read symbol (encoded in a flag), and the next state. An entry in the read state portion of the check table looks as follows:

expected symbol	flag	next state

In general a semantic stack runs in parallel with the syntactic stack. When a "read" symbol was encountered then semantic information is also pushed onto the semantic stack, e.g. the variant number of the terminal is saved. Depending on the organization of the semantic stack, the "value" of a lexeme, or a pointer to the value of a lexeme is also stored on the semantic stack.

If the input symbol is not among the expected symbols then the action associated with the read state is carried out. If the read state has a default transition then the parser goes to the indicated next state, otherwise an error has been encountered and an error message is issued. The read state also contains information on how many entries for this state are in the check table, and where to find them. A read state entry looks as follows:

index of check table	number of entries	flag	error number or next state

The flag indicates if the read state contains a default transition or an error number.

So far it was assumed that the check entries of a read state are stored consecutively. In order to save space, however, check entries of several read states are overlapped. The different parts of the check entries of a read state are linked together by link entries. A link entry looks as follows:

0	index of check table

A link entry is at the end of a block of check entries. When during the search of the expected symbols the zero entry is encountered then the search is continued at the indicated place in the check table.

Reduce State

In a reduce state the parser pops the appropriate number of elements off the stack and performs a semantic action. Semantic information which corresponds to the symbols of the righthand side of the production is accessible with

stacktop+0, stacktop+1, ..., stacktop+length-1

on the semantic stack. Length is the number of symbols on the righthand side of the production. The results of a semantic action are stored at stacktop+0 on the semantic stack. (Note this also takes care of empty productions). After the reduction the parser goes to the next state given by the reduce state. A reduce state entry looks as follows:

semantic action number	length of production	next state

Lookback State

In a lookback state the parser "reads" the state on top of the stack and decides which state to go to. A lookback state could be organized in the same manner as a read state by using state numbers instead of symbol numbers.

However, a more efficient structure avoids any searches. The state number is added to an offset provided by the lookback state and the resulting value is used to index the check table. If the appropriate location is indexed then the parser goes to the next state which is found at this location, otherwise the parser goes to the next state which is associated with the lookback state. A lookback state entry looks as follows:

offset	next state

The same place in the check table may be indexed by different offset/state number combinations. In order to know if the computed index is appropriate also the state number of the lookback state for which the check entry is to be valid is stored together with the next state information. An entry in the lookback state portion of the check table looks as follows:

lookback state number	next state

Only if the current lookback state is the same as the recorded lookback state then a proper index has been computed and the parser goes to the indicated next state.

4.2 Error Recovery

When an error is encountered then the error recovery mechanism is invoked. Input symbols and/or stack symbols are deleted until a state and an input symbol is found to continue parsing. The recovery mechanism works in 4 steps.

Step 1: Insertion

It is assumed that a symbol was omitted in the input. The parser inserts one of the expected symbols of the current state and checks if it can continue with the input symbol at hand. This is tried out for all

symbols in this state, and if successful the first one which allows the parser to continue is chosen.

Step 2: Deletion

The current input symbol is discarded and a the next input symbol is obtained. The parser checks if parsing can continue with this symbol.

Step 3: Replacement

The same actions are carried out as in step 1, this time with the new input symbol, however.

Step 4: Panic Mode

The parse stack is searched for a state in which parsing can continue with the input symbol at hand. If no such state is found then the current symbol is discarded, a new input symbol is obtained, and step 4 is repeated.

4.3 Error Recovery by Error Productions

A different error recovery mechanism is available if the grammar of the language is formulated with "error" productions. A lexeme ERROR may be used for this purpose.

The terminal ERROR does not appear in the input stream, however, when an error is detected then the unread portion of the input is prefixed by the ERROR terminal. The stack is searched for a state which has a transition on the ERROR symbol. The following read state provides a set of expected symbols. Input is skipped until a symbol is found which is among the expected symbols. Parsing then continues. At least two more symbols must be read from input without causing any new errors before another error is reported. Error productions may be associated with semantic action numbers.

The recovery behaviour is demonstrated with the following productions.

```
/DECLARATION/ = VAR /NAME LIST/ ;  $
                ERROR ;   !
```

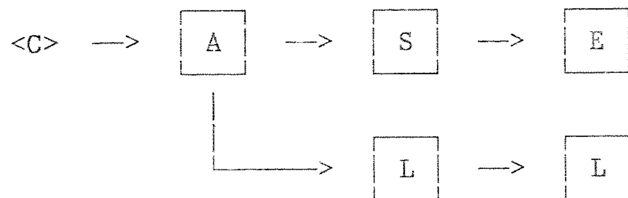
Whenever an error occurs in the name list then the parser backs up to recognizing a declaration as this state has an error transition. Input is skipped until a semicolon is found.

Enough error productions must be supplied by the user so that there is always some state on the stack which has a useful error transition. Poonen [11] suggests a similar error recovery mechanism.

5. Lexical Scanner

Lexical scanner tables are generated from the tokens of the language when the output of the parse table generator is transformed for a particular skeleton compiler. There are two basic structures which are generated depending on the skeleton compiler.

The first structure to be handled is a forest of trees. In essence it is a deterministic finite automaton. A table which is indexed by the internal representation of the first character of a token contains the roots of the trees. For example the terminal symbols CASE and CALL are stored as follows:



A token is recognized by following the links of the tree according to the characters read from input. A final state in the tree contains the internal number and the variant number of the recognized terminal. These numbers are returned to the parser when the lexical scanner is called to obtain the next token from the input string.

The second organization is an indexed-sequential structure. Reserved words (tokens consisting of letters) are grouped according to their length. A directory is made in order to find a group of a particular length. An input symbol is recognized by searching the group of tokens with the appropriate length. For example the following storage structure is used to store the terminals FOR, BEGIN, END:

directory

1	0	0
2	0	0
3	1	2
4	0	0
5	3	1
	etc.	

token table

1	FOR
2	END
3	BEGIN

The first column of the directory gives the index where a group of tokens starts, the second column contains the number of entries in this group.

Tokens consisting of special symbols are treated separately. They may be stored as trees or in token tables.

The standard lexical scanner also recognizes certain lexemes, namely identifiers, numbers, and strings. In addition comments are recognized. The "value" of a lexeme may be also returned to the parser. However, depending on the language which is to be implemented, special processing

may be required, e.g. strings might be stored in a string table after they are recognized, or identifiers might be entered into a hash table. The appropriate modifications of the standard lexical scanner must be made by the user.

6. Skeleton Compiler

Skeleton compilers are available in several languages. They are available in Pascal, Fortran, Lisp, Simula and 8080 Assembly. A skeleton compiler consists of a "parser" part and a "semantic" part. The parser part consists of input/output routines, lexical scanner, and parsing algorithm with error recovery. The semantic part consists of an interface to the parser. The user must provide the routines and data structures to implement the semantic actions.

The lexical scanner tables and parse tables are global structures. If the language in which the skeleton compiler is written allows data statements, then the generated tables are inserted as initializations into the source of the skeleton compiler by the table transformation program. In other cases the generated tables are read from a file before compilation begins.

Some interface variables are provided to make it easier to adapt the skeleton compiler to a particular language. In the following special attention is given to the skeleton compiler written in Pascal. However, all features described are also available in a similar way in the other skeleton compilers.

6.1 Parser Interface

The internal values of lexemes are stored in an array LEXVAL. They are interfaced to the lexical scanner as follows:

```
NAMVAL     := LEXVAL[1];
KONSTVAL   := LEXVAL[2];
STRINGVAL  := LEXVAL[3];
COMMENTVAL := LEXVAL[4];
```

If the error recovery mechanism by error productions is used then the interface must be changed to:

```
ERRORVAL   := LEXVAL[4];
COMMENTVAL := LEXVAL[5];
```

Additional lexemes in the language require modifications to the lexical scanner. If fewer lexemes are used then the value zero should be assigned to the internal variable for which no lexeme is provided.

The character set is grouped into 4 categories: letters, digits, special characters and delimiters. The array KONV contains the group property of each character. A character is placed into a different category by changing its property. This e.g. may be required when option 5 is used. The accepted character set depends on machine and language. The skeleton compiler in Pascal for example handles a 63 or 64 character set on the CDC 6600. The skeleton compiler in Pascal on the DEC-10 handles a 96 character set.

The parser is called with one parameter. This parameter indicates which goal symbol is to be used. The main goal symbol has number 1. All other goal symbols are numbered in increasing order according to the sequence in which they are listed in the ENTRIES section.

All standard parsers provide the error recovery mechanism described in section 4.2. The user may, however, choose the error recovery mechanism by error productions. This error recovery mechanism, for example, is included as comment in the Pascal skeleton compiler.

6.2 Semantic Interface

The interface between parser part and semantic part is established with the routine SEMANTICS and a semantic stack. The routine SEMANTICS is called from the parser each time a reduction is performed (and a semantic action is associated with the production). The semantic stack runs in parallel with the syntactic stack. The variable STKTOP points to the stack entry which corresponds the the first symbol on the righthand side of the respective production when the routine is called. Other parts of the righthand side are reached with the proper offset to STKTOP.

The structure of an object on the semantic stack depends on the language and on the design of other semantic structures, like the symbol table. The lexical scanner delivers the "value" of a token on the semantic stack. For lexemes the value for example may be the character string which represents an identifier, or the value may be a pointer to a structure where the identifier is stored. For tokens the value is the token itself. The type of an object on the semantic stack and the semantic stack is defined in the Pascal parser by:

```
TYPE
   OBJTY = RECORD
             CASE N:INTEGER OF
           1: (TOK: INTEGER);    (* TOKEN *)
           2: (NAM: ALFA);       (* IDENTIFIER *)
           3: (INM: INTEGER);    (* INTEGER NUMBER *)
           4: (RNM: REAL);       (* REAL NUMBER *)
           5: (STP: INTEGER);    (* POINTER TO STRING *)
           6: (PTR: INTEGER);
           END;

VAR
   SEM: ARRAY[0..STKMAX] OF OBJTY;
```

The user must extend the type definition according to the objects which will be stored on the semantic stack (for example the field PTR may be used for pointers).

Syntax classes are handled in a similar way. The lexical scanner stores the variant number of a terminal on the stack. In the Pascal parser the variant number is stored on the syntax stack in the field CHOICE. The variant number of a token for example which is stored at STKTOP can be made available to a semantic action by STK[STKTOP].CHOICE.

The routine SEMANTICS is a case statement which implements the semantic actions. The semantic action number is used to select the appropriate case. Semantic actions work in general by side effects, e.g. they manipulate data structures like the symbol table. A semantic action may also return a value. The value must be stored on top of the semantic stack. (The top of the stack corresponds to the left side of the production).

We consider now the following productions and their semantic actions.

```
/FACTOR/ = IDENTIFIER $2
           ( /EXPRESSION/ ) !3
/EXPRESSION/ = /EXPRESSION/ /OP/ /EXPRESSION/ $4
               /FACTOR/ !
```

The semantic action for the first production requires the identifier to be looked up in a symbol table. We return a pointer to the place where the symbol is stored (this e.g. can take care of an undefined symbol by returning zero). The second production moves the semantic contents to the top of the stack. The third production builds a prefix form and stores a pointer to it on the stack.

A Pascal implementation might look as follows (the variable PNR contains the semantic action number when the semantic routine is called):

```
PROCEDURE SEMANTICS;
BEGIN
  CASE PNR OF
 2: WITH SEM[STKTOP] DO BEGIN
      PTR:=CHECKSYMBOL(TOK);
      N:=6;
    END;
 3: SEM[STKTOP]:=SEM[STKTOP+1];
 4: WITH SEM[STKTOP] DO BEGIN
      PTR:=MKPREFIX(1,0,2);
      N:=6;
    END;
    ...

END;
```

The routines CHECKSYMBOL and MKPREFIX must be provided by the user.

In general semantic actions must be coded by the user. The skeleton compiler in Lisp provides a somewhat more interesting way for specifying semantic actions. A file is supplied which associates semantic action numbers with Lisp functions. The file for the productions above looks as follows:

```
2  (CHECKSYMBOL 0)
3  1
4  (LIST 1 0 2)
```

When the file is read the numbers are transformed into the appropriate index operations for the semantic stack. Each semantic action returns a value on top of the stack. Semantic actions in this form can be quickly prepared, e.g. to make a translator for transforming programs into some prefix or postfix form.

6.3 Interactive Usage

Interactive use of the parser requires some special considerations. The main problem is how the 'end-of-file' token is obtained in order to terminate a sentence which reduces to a goal symbol of the language. Example:

```
entries /option/ /process/ !
  /option/ = number $
             /option/ , number !
  /process/ = string !
```

In many cases the end-of-line has the function of terminating the sentence, or there is an obvious last terminal with this role.

In order to solve this problem interaction between parser and lexical scanner is necessary. A special skeleton compiler is provided which demonstrates the basic mechanism. The lexical analyser is divided into a 'screener' and a token recognizer. Depending on context an end-of-line condition can become significant and returned as a 'end-of-file' token. Option 23 must be used with the XBSW system to retain the complete lookahead information for states which contain the 'end-of-file' token.

The above grammar rules could be used in conjunction with a command parser. The following is an example of a possible user dialog.

```
->  option 1
->  option 1, 2,
P>    3
->  process   'abc'
->  process
P>     'abc'
->
```

The prompt characters from the command processor are '->', the prompt characters from the parser are 'P>'. The command key words 'option' and 'process' are not part of the language; they are dealt with independently by the command processor. (This skeleton compiler for interactive parsing is available for DEC-20, IBM-CMS, and VAX environments).

7. Some Results

The grammars of two modern languages Pascal and C [6,12] were used to produce parsers. The grammar for Pascal consists of 159 productions, the grammar for C of 132 productions. (The grammar for C was taken directly from the Unix manual). The following table gives the number of entries in the state and check table for these grammars.

	Pascal	C	
State table	201	178	read
	22	18	lookback
	154	133	reduce
Total:	377	329	
Check table	232	229	read
	84	95	lookback
Total:	316	324	

The following table gives the storage requirments (in 36 bit words) for lexical scanner tables and parse tables for the Pascal parser program on the DEC-10. (The lexical scanner does not include keywords for the C preprocessor).

	Pascal	C
lexical scanner	294	295
parser	693	653
Total:	987	948

The following table gives the storage requirements in bytes for a parser on an 8-bit machine. It is noteworthy that the tables for Pascal and C occupy only about 2K bytes.

	Pascal	C
lexical scanner	327	345
parser	1741	1617
Total:	2068	1962

A complete skeleton compiler was written for the 8080 micro-processor in assembly language. The lexical scanner algorithm requires about 800 bytes, the parsing algorithm with error recovery requires about 600 bytes.

References

[1] DeRemer, F.L.,
Simple LR(k) Grammars.
CACM, 14,7, July 1971.

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

[3] Aho, A.V., Johnson, S.C., and Ullman, J,D.,
Deterministic Parsing of Ambiguous Grammars.
ACM Symposium on Principles of Programming Languages, Boston, October 1973.

[4] Aho, A.V., and Ullman, J.D.,
A Technique for Speeding up LR(k) Parsers.
SIAM, 2,2, June 1973.

[5] Aho, A.V., and Ullman, J.D.,
Principles of Compiler Design.
Addison-Wesley, 1977

[6] Jensen, K. and Wirth, N.,
PASCAL User Manual and Report.
Springer Verlag, 1975.

[7] Burger, W.F.,
BOBSW 3.0 - A Parser Generator.
TR-87, Department of Computer Sciences, University of Texas at Austin, November 1978.

[8] Thompson, D.H.,
The Design and Implementation of an Advanced LALR Parse Table Constructor.
CSRG-79, Computer Systems Research Group, University of Toronto, April 1977.

[9] Burger, W.F.,
Parser Generation for Micro-Computers.
TR-77, Department of Computer Sciences, University of Texas at Austin, March 1978.

[10] Burger, W.F.,
A Parser Generation Tool for Micro-Computers.
COMPSAC, November 1979.

[11] Poonen, G.,
Error Recovery for LR(k) Parsers.
IFIP 1977.

[12] Ritchie, D.M., et al.,
The C Programming Language.

CSTR-31, Bell Laboratories, Murray Hill, N.J., 1975.

[13] Ambler, A., Good, D.I., Burger, W.F.,
Report on the Language Gypsy.
ICSCA-CMP-1, University of Texas at Austin, August 1976.

[14] Hilfinger, P.,
Private Communication, November 1978.

[15] Intermetrics, Inc.
REDL - Informal Language Specification.
February 1978.

[16] Brosgol, B.,
Private Communication, August 1979.

[17] Browne, J.C., Johnson, D.B.,
FAST: A Second Generation Program Analysis System.
3rd International Conference on Software Engineering, Atlanta, 1978.

[18] Elliot, L., Kunii, H., Browne, J.C.,
A Data Management System for Engineering and Scientific Computing.
Proceedings on Engineering and Scientific Data Management, NASA, May 1978.

[19] Carl-Mitchell, F.S., Berry, R.F., Dale, A.G.,
APRIL: Accessing and Processing Interface Language for Database Applications.
ICSCA-RDS-2, University of Texas at Austin, February 1979.

[20] Meeraus, A.,
General Algebraic Modeling System.
The World Bank, August 1977.

Appendix A

Options (Part 1)

These options are used in the OPTIONS section of the input to the table generator.

General

1 The grammar is treated as LALR(1) grammar instead of SLR(1) grammar.

2 The grammar may be ambiguous.

3 An attempt is made to fit the pretty-printed grammar into 80 columns.

Grammar Input

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

5 Terminal symbols may consist of characters of both character groups. (The character classification in the standard lexical scanners must be adapted).

6 The internal encoding of terminals and nonterminals is printed.

7 The bitmatrix of the grammar is printed.

8 Empty productions are removed from the grammar.

Listing

10 Suppress listing of input.

11 Suppress listing of grammar checks.

13 Suppress printout of conflict resolution of inadequate states by disambiguation sets.

14 Suppress printout of conflict resolution of inadequate states by precedence rules.

LR(0) Generation

15 The states of the LR(0) machine are printed. The state of the parse is shown together with the production involved. (Caution: this option generates a large amount of output).

16 The internal form of the LR(0) machine is printed.

17 The internal form of lookback states is printed.

18 The SLR(1) lookahead symbols for each nonterminal are printed.

19 The inadequate states of the LR(0) machine are printed.

Optimizer

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

21 States with the same tail are not folded together.

22 No attempt is made to remove simple productions.

23 Lookahead sets are not removed from states which contain the -EOF- token. (This option is useful in conjunction with grammar entries and interactive parsing).

24 The final form of the parse tables (produced by the optimization module) is printed.

Miscellaneous

26 The user supplied semantic action numbers are associated with the productions.

29 Metasymbol 5 is enabled.

Options (Part 2)

These options are used on the control card when executing part 2 of the table generator.

S An attempt is made to overlap as many read check entries as possible in order to save space.

M An attempt is made to keep offset values small when the lookback states are transformed. This option is only required when tables for a micro-computer are generated.

L A full listing of the final tables is produced.

With some systems it is not possible to pass options on the control card to the program. In this case the desired options must be already provided in part 1. The option numbers are:

30 same as S
31 same as M
32 same as L

Appendix B

Constants of the Parse Table Generator

The following constants are used to define table sizes and record fields of the parse table generator. The values of a typical system are given. The system must be re-compiled if any of these constants needs to be changed.

CONST1	400	Number of productions.
CONST2	300	Number of terminal and nonterminal symbols.
CONST3	50	Table for counters used by optimizer.
CONST4	8000	Array for rightside of productions.
CONST5	7	Maximum length of right side of a production.
CONST6	12000	Array for parser table.
CONST7	63	Maximum size of a syntax class.
CONST8	100	Table for lookahead symbols.
CONST9	500	Stack for configuration sets.
CONST10	1023	Table for parser states.
CONST11	200	Array for nonterminals.
CONST12	500	Array for terminal tree.
CONST14	15	Stack used to compute lookahead sets.
CONST15	63	Maximum size of a basis set.
CONST16	2000	Table for LALR(1) lookahead.
CONST17	30	Maximum number of syntax classes.
CONST18	31	Table for lexemes and other keywords.
CONST19	400	Table for symbols of disambiguation sets.
CONST20	200	Table used to eliminate simple productions.
CONST21	100	Table used to find lookahead-error states.
CONST24	10	Maximum number of goal symbols.
CONST27	20	Maximum length of a terminal symbol.

Appendix C

The following pages contain a complete listing of a parse table generation. The grammar is taken from [5] and slightly modified. It is a subset of Pascal. The grammar is formulated with syntax classes, disambiguation sets and precedence rules. Semantic action numbers are assigned by the system.

The generated parse tables are used with the Pascal parser program on the DEC-10. A sample program is translated with the generated parser. The sequence of semantic action calls is recorded.

C.1 Parse Table Generation

```
 ****************  LISTING OF INPUT 27-Nov-79 11:10:12    ****************

  1  OPTIONS 1,2,3,14
  2
  3  *
  4  *  A PASCAL SUBSET
  5  *  (FROM AHO AND ULLMAN: PRINCIPLES OF COMPILER DESIGN)
  6  *
  7
  8  EMPTY EMPTY
  9
 10  CLASSES
 11  + + -
 12  + <> <= >= < >
 13  + * DIV MOD / AND
 14  + INTEGER REAL
 15
 16  TERMINALS
 17    .. . [ ] ( ) ; , : = :=
 18    PROGRAM VAR ARRAY OF FUNCTION
 19    PROCEDURE BEGIN END IF THEN ELSE WHILE DO NOT OR
 20
 21  LEXEMES
 22    IDENTIFIER NUMBER STRING
 23
 24  SCANNER
 25    (*
 26
 27  PRECEDENCE
 28  BINARY = <>
 29  LEFT + OR
 30  LEFT *
 31
 32  STRINGCH "
 33  ENDCH *)
 34
 35  PRODUCTIONS
 36  METASYMBOLS M1=# M2=$ M3=/ M4=!
 37
 38    /PROGRAM/ # /PROGRAM HEADER/ /PROGRAM BODY/ !
 39    /PROGRAM HEADER/ # PROGRAM IDENTIFIER ( /IDENTIFIER LIST/ ) ; !
 40    /PROGRAM BODY/ # /DECLARATIONS/ /COMPOUND STATEMENT/ . !
 41    /IDENTIFIER LIST/ # IDENTIFIER $
 42                        /IDENTIFIER LIST/ , IDENTIFIER !
 43    /DECLARATIONS/ # /VAR DECLARATION/ /ROUTINE DECLARATIONS/ !
 44    /VAR DECLARATION/ # EMPTY $
 45                        VAR /DECLARATION LIST/ !
 46    /DECLARATION LIST/ # /IDENTIFIER LIST/ : /TYPE/ ; $
 47                    /DECLARATION LIST/ /IDENTIFIER LIST/ : /TYPE/ ; !
```

```
48    /TYPE/ # /STANDARD TYPE/ $
49             /ARRAY TYPE/ !
50    /STANDARD TYPE/ # INTEGER !
51    /ARRAY TYPE/ # ARRAY [ /ARRAY INDEX/ ] OF /STANDARD TYPE/ !
52    /ARRAY INDEX/ # /CONSTANT/ .. /CONSTANT/ !
53    /ROUTINE DECLARATIONS/ # EMPTY $
54                        /ROUTINE DECLARATIONS/ /ROUTINE DECLARATION/ !
55    /ROUTINE DECLARATION/ # /ROUTINE HEADER/ /DECLARATIONS/
56                              /COMPOUND STATEMENT/ ; !
57    /ROUTINE HEADER/ # PROCEDURE IDENTIFIER /ARGUMENTS/ ; $
58                 FUNCTION IDENTIFIER /ARGUMENTS/ : /STANDARD TYPE/ ; !
59    /ARGUMENTS/ # ( /PARAMETER LIST/ ) $
60                   EMPTY !
61    /PARAMETER LIST/ # /IDENTIFIER LIST/ : /TYPE/ $
62                       /PARAMETER LIST/ ; /IDENTIFIER LIST/ : /TYPE/ !
63    /COMPOUND STATEMENT/ # BEGIN /STATEMENT LIST/ END !
64    /STATEMENT LIST/ # /STATEMENT/ $
65                          /STATEMENT LIST/ ; /STATEMENT/ !
66    /STATEMENT/ # /VARIABLE/ := /EXPRESSION/ $
67                  /PROCEDURE STATEMENT/ $
68                  /COMPOUND STATEMENT/ $
69                  IF /EXPRESSION/ THEN /STATEMENT/ ELSE /STATEMENT/ $
70                  IF /EXPRESSION/ THEN /STATEMENT/ $
71  C ELSE
72                  WHILE /EXPRESSION/ DO /STATEMENT/ $
73                  EMPTY  !
74    /VARIABLE/ # IDENTIFIER $
75                 IDENTIFIER [ /EXPRESSION/ ] !
76    /PROCEDURE STATEMENT/ # IDENTIFIER $
77                            IDENTIFIER ( /EXPRESSION LIST/ ) !
78    /EXPRESSION LIST/ # /EXPRESSION/ $
79                         /EXPRESSION LIST/ , /EXPRESSION/ !
80    /EXPRESSION/ # /EXPRESSION/ = /EXPRESSION/ $
81                   /EXPRESSION/ <> /EXPRESSION/ $
82                   /EXPRESSION/ + /EXPRESSION/ $
83                   /EXPRESSION/ OR /EXPRESSION/ $
84                   /EXPRESSION/ * /EXPRESSION/ $
85                   /FACTOR/ !
86    /FACTOR/ # /VARIABLE/ $
87               NUMBER $
88               STRING $
89               ( /EXPRESSION/ ) $
90               IDENTIFIER ( /EXPRESSION LIST/ ) $
91               + /FACTOR/ $
92               NOT /FACTOR/ !
93
94    /CONSTANT/ # + NUMBER $
95                 NUMBER !
96  !
```

****************************** END OF LISTING ******************************

****************************** GRAMMAR CHECKS ******************************

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

IT HAS BEEN CHECKED THAT THERE
EXIST NO IDENTICAL PRODUCTIONS

THE GRAMMAR HAS BEEN CHECKED FOR
SIMPLE CHAINS

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

<EXPRESSION>

THE NONTERMINALS ABOVE ARE BOTH
LEFT AND RIGHT RECURSIVE

```
******************** THE GRAMMAR 27-Nov-79 11:10:12 ********************

 1      <PROGRAM> ::= <PROGRAM HEADER> <PROGRAM BODY>

 2      <PROGRAM HEADER> ::= PROGRAM IDENTIFIER (
                              <IDENTIFIER LIST> ) ;

 3      <PROGRAM BODY> ::= <DECLARATIONS> <COMPOUND STATEMENT> .

 4      <IDENTIFIER LIST> ::= IDENTIFIER
 5                          / <IDENTIFIER LIST> , IDENTIFIER

 6      <DECLARATIONS> ::= <VAR DECLARATION>
                          <ROUTINE DECLARATIONS>

 7      <VAR DECLARATION> ::= EMPTY
 8                          / VAR <DECLARATION LIST>

 9      <DECLARATION LIST> ::= <IDENTIFIER LIST> : <TYPE> ;
10                           / <DECLARATION LIST>
                                <IDENTIFIER LIST> : <TYPE> ;

11      <TYPE> ::= <STANDARD TYPE>
12               / <ARRAY TYPE>

13  0   <STANDARD TYPE> ::= INTEGER
    1                     / REAL

14      <ARRAY TYPE> ::= ARRAY [ <ARRAY INDEX> ] OF
                           <STANDARD TYPE>

15      <ARRAY INDEX> ::= <CONSTANT> .. <CONSTANT>

16      <ROUTINE DECLARATIONS> ::= EMPTY
17                               / <ROUTINE DECLARATIONS>
                                   <ROUTINE DECLARATION>

18      <ROUTINE DECLARATION> ::= <ROUTINE HEADER> <DECLARATIONS>
                                   <COMPOUND STATEMENT> ;

19      <ROUTINE HEADER> ::= PROCEDURE IDENTIFIER <ARGUMENTS> ;
20                         / FUNCTION IDENTIFIER <ARGUMENTS> :
                              <STANDARD TYPE> ;

21      <ARGUMENTS> ::= ( <PARAMETER LIST> )
22                    / EMPTY

23      <PARAMETER LIST> ::= <IDENTIFIER LIST> : <TYPE>
24                         / <PARAMETER LIST> ; <IDENTIFIER LIST> :
                              <TYPE>

25      <COMPOUND STATEMENT> ::= BEGIN <STATEMENT LIST> END
```

```
26       <STATEMENT LIST> ::= <STATEMENT>
27                          / <STATEMENT LIST> ; <STATEMENT>

28       <STATEMENT> ::= <VARIABLE> := <EXPRESSION>
29                     / <PROCEDURE STATEMENT>
30                     / <COMPOUND STATEMENT>
31                     / IF <EXPRESSION> THEN <STATEMENT> ELSE
                          <STATEMENT>
32                     / IF <EXPRESSION> THEN <STATEMENT>
33                     / WHILE <EXPRESSION> DO <STATEMENT>
34                     / EMPTY

35       <VARIABLE> ::= IDENTIFIER
36                    / IDENTIFIER [ <EXPRESSION> ]

37       <PROCEDURE STATEMENT> ::= IDENTIFIER
38                               / IDENTIFIER ( <EXPRESSION LIST> )

39       <EXPRESSION LIST> ::= <EXPRESSION>
40                           / <EXPRESSION LIST> , <EXPRESSION>

41       <EXPRESSION> ::= <EXPRESSION> = <EXPRESSION>
42   0                  / <EXPRESSION> <> <EXPRESSION>
     1                  / <EXPRESSION> <= <EXPRESSION>
     2                  / <EXPRESSION> >= <EXPRESSION>
     3                  / <EXPRESSION> < <EXPRESSION>
     4                  / <EXPRESSION> > <EXPRESSION>
43   0                  / <EXPRESSION> + <EXPRESSION>
     1                  / <EXPRESSION> - <EXPRESSION>
44                      / <EXPRESSION> OR <EXPRESSION>
45   0                  / <EXPRESSION> * <EXPRESSION>
     1                  / <EXPRESSION> DIV <EXPRESSION>
     2                  / <EXPRESSION> MOD <EXPRESSION>
     3                  / <EXPRESSION> / <EXPRESSION>
     4                  / <EXPRESSION> AND <EXPRESSION>
46                      / <FACTOR>

47       <FACTOR> ::= <VARIABLE>
48                  / NUMBER
49                  / STRING
50                  / ( <EXPRESSION> )
51                  / IDENTIFIER ( <EXPRESSION LIST> )
52   0              / + <FACTOR>
     1              / - <FACTOR>
53                  / NOT <FACTOR>

54   0  <CONSTANT> ::= + NUMBER
     1               / - NUMBER
55                   / NUMBER
```

****

```
INADEQUATE STATES ARE TO BE RESOLVED WITH THE FOLLOWING SETS:

PRODUCTION   KIND

   32        CONTINUE  ELSE

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

PRODUCTIONS WITH ASSOCIATED PRECEDENCE:

       41          1           42          1           43          2
       44          2           45          3           46          3
       52          2

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


<ARGUMENTS>                             19      20      21 L
<ARRAY INDEX>                           14      15 L
<ARRAY TYPE>                            12      14 L
<COMPOUND STATEMENT>                     3      18      25 L    30
<CONSTANT>                              15      15      54 L
<DECLARATION LIST>                       8       9 L    10
<DECLARATIONS>                           3       6 L    18
<EXPRESSION>                            28      31      32      33      36
                                        39      40      41 L    41      41
                                        42      42      43      43      44
                                        44      45      45      50
<EXPRESSION LIST>                       38      39 L    40      51
<FACTOR>                                46      47 L    52      53
<IDENTIFIER LIST>                        2       4 L     5       9      10
                                        23      24
<PARAMETER LIST>                        21      23 L    24
<PROCEDURE STATEMENT>                   29      37 L
<PROGRAM>                                1 L
<PROGRAM BODY>                           1       3 L
<PROGRAM HEADER>                         1       2 L
<ROUTINE DECLARATION>                   17      18 L
<ROUTINE DECLARATIONS>                   6      16 L    17
<ROUTINE HEADER>                        18      19 L
<STANDARD TYPE>                         11      13 L    14      20
<STATEMENT>                             26      27      28 L    31      31
                                        32      33
<STATEMENT LIST>                        25      26 L    27
<TYPE>                                   9      10      11 L    23      24
<VAR DECLARATION>                        6       7 L
<VARIABLE>                              28      35 L    47
```

```
***********************  INADEQUATE STATES (LALR1)  ***********************

   STATE        PRODUCTION  PLACE        SYMBOL

     108  ---------------------------------------------------------------

          READ          31       5         ELSE

          REDUCTION     32
                          ;  END  ELSE

          THE CONFLICT IS RESOLVED AS FOLLOWS:

          CONTINUE        ELSE

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

         THE GRAMMAR IS LALR1
         THE AMBIGUITIES OF THE GRAMMAR COULD BE RESOLVED

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

*************  COMPILER ERROR MESSAGES 27-Nov-79 11:10:12   *************

  ERRORNO :          EXPECTED SYMBOL:

        0 :          ** SPECIAL ERROR **

        1 :          PROGRAM

        2 :          IDENTIFIER

        3 :          (

        4 :          )  ,

        5 :          ;

        6 :          :  ,

        7 :          INTEGER  ARRAY

        8 :          [

        9 :          +  NUMBER

       10 :          NUMBER

       11 :          ..

       12 :          ]

       13 :          OF
```

```
14 :        INTEGER

15 :        )  ;

16 :        :

17 :        BEGIN

18 :        NUMBER  STRING  (  IDENTIFIER  +  NOT

19 :        )  =  <>  +  OR  *

20 :        ]  )  ;  ,  END  THEN  ELSE  DO  +  OR  *

21 :        ]  =  <>  +  OR  *

22 :        DO  =  <>  +  OR  *

23 :        THEN  =  <>  +  OR  *

24 :        :=

25 :        END  ;

26 :        .

27 :        -EOF-
```

****

STORAGE REQUIREMENTS

CONST	ALLOCATED	USED	
1	400	56	NUMBER OF PRODUCTIONS
2	300	74	TERMINALS AND NONTERMINALS
3	50	12	TABLE FOR FREQUENCY COUNTERS
4	8000	565	ARRAY SIZE FOR RIGHTSIDE OF PRODUCTIONS
6	12000	234	ARRAY SIZE FOR PARSER TABLES
		767	ACTUALLY USED
8	100	13	TABLE FOR LOOKAHEAD SYMBOLS
9	500	59	STACK USED TO GENERATE CONFIGURATION SETS
10	1023	168	TABLE FOR PARSER STATES
11	200	25	ARRAY SIZE FOR NONTERMINALS
12	500	68	ARRAY SIZE FOR TERMINAL TREE
		91	ACTUALLY USED
16	2000	345	TABLE FOR LALR(1) LOOKAHEAD SYMBOLS
19	400	2	TABLE FOR DISAMBIGUATION SYMBOLS
21	100	24	TABLE USED TO FIND LOOKAHEAD-ERROR STATES

TIME USED   9975 MSEC

THE SEMANTIC LABELS ARE SYSTEM GENERATED.

C.2 Parse Table Transformation

```
 TRANSFORMATION OF PARSE TABLES                           27-Nov-79 11:14:54


 OPTIONS: 'S' SPACE OPTIMIZATION    NO
          'M' MICRO HEURISTIC       NO
          'L' LISTING               NO

 (TABLES OF 27-NOV-79 11:10:12  )


 ALL STATES:            154
 (READ)                  86
 (LOOKBACK)              12
 (REDUCE)                56

 CHECK TABLES:          134
 (READ)                  97
 (LOOKBACK)              37

 TIME USED          2654 MSEC
```

C.3 Parser Program Generation

```
 THE TABLES WERE GENERATED FOR DEC-10 PASCAL      27-Nov-79 11:16:08

 (TABLES OF 27-Nov-79 11:14:54  )

 STATE TABLE             154 WORDS
 CHECK TABLE             134 WORDS
 TERMINAL TREE           206 WORDS
 INTERNREP                35 WORDS

 LEXEMES AND OTHER LEXICAL ITEMS

          32   IDENTIFIER
          33   NUMBER
          34   STRING
          35   (*

 VERSION INFO:  XBSW 3.3
```

C.4 Example Program

```
        COMPILER VERS NOV-79        27-Nov-79 11:19:49
        XBSW 3.3

    1   program example(input,output);
    2   var x,y:integer;
    3   function gcd(a,b:integer):integer;
    4   begin
    5     if b=0 then gcd:=a else gcd:=gcd(b, a mod b)
    6   end;
    7   begin
    8     read(x,y);
    9     write(gcd(x,y));
   10   end.

        NO ERRORS IN PROGRAM

        COMPILE TIME       0.128 SEC.
```

The following calls to semantic actions are made during the translation of this program:

```
  4   5   2   4   5  13  11   9   8  16   4   5
 13  11  23  21  13  20   7  16   6  35  47  46
 48  46  41  35  35  47  46  28  35  35  47  46
 39  35  47  46  35  47  46  45  40  51  46  28
 31  26  25  18  17   6  35  47  46  39  35  47
 46  40  38  29  26  35  47  46  39  35  47  46
 40  51  46  39  38  29  27  34  27  25   3   1
```

Appendix D

D.1 Parse Table Generation on DEC-10 and DEC-20

For historic reasons the table generator of the XBSW system consists of two parts. Part 1 contains the first three modules. Part 2 contains the rest of the modules. The two parts must be executed in sequence.

It is assumed that the grammar is on file LAN.GRM. The command sequence to obtain parse tables PARS.TAB is as follows:

```
RUN XBSW
GRAMMAR  = LAN.GRM
OUTPUT   = LAN.LST
TMPTAB   = TMPTAB
; Intermediate tables are generated on file TMPTAB,
; a listing is produced on file LAN.LST

RUN XTRANS
TMPTAB   = TMPTAB/S
OUTPUT   = TRNS.LST
TABLES   = PARS.TAB
```

The program XTRANS can be used with the options S, M, and L (see appendix A).

D.2 Pascal Skeleton Compiler for DEC-10

The tables obtained with the table generator are transformed with the program XPGEN for the DEC-10 Pascal skeleton compiler. It is assumed that the output of the table generator is on file PARS.TAB. Any place holder terminals must be substituted by their actual terminals before this file is further processed (see section 3.1). This can be done, for example, with a text editor.

The tables PARS.TAB are transformed into DEC-10 Pascal "initprocedures". The initprocedures may be inserted into the template file XPRS.PAS which is the standard skeleton compiler.

```
RUN XPGEN
TABLES  = PARS.TAB
INSERT TABLES INTO A TEMPLATE FILE (Y OR N) ?  Y
FILE    = XPRS.PAS
; The parser program is generated on file PARS.PAS,
; output is sent to file PARS.OUT.
```

The resulting program can be translated with the Pascal compiler and executed. The following example assumes that a program TEST is available.

```
EXEC PARS.PAS
....
SOURCE  = TEST/P
; A listing is produced on file TEST.LST,
; the semantic actions called are displayed on the
; terminal.
```

The options of the parser program are:

```
N  No program listing is produced.
U  An input line is 80 characters long instead of 72.
P  Display the semantic action number when the semantic
   routine is called.
```

D.3 Pascal Skeleton Compiler for DEC-20

The tables obtained with the table generator are transformed with the program XGEN for the DEC-20 Pascal skeleton compiler. The tables are either transformed into DEC-20 Pascal "initprocedures" or when option P is used into assignment statements. The latter form enhances the transportability of the generated parser.

```
RUN XGEN
TABLES  = PARS.TAB/P
PARSIN  = XPRS.PAS
NEWPRS  = PARS.PAS
OUTPUT  = PARS.OUT
;  The parser program is generated on file PARS.PAS,
;  output is sent to file PARS.OUT
```

The generated program can be translated with the Pascal compiler and executed. The following example assumes that a program TEST is available.

```
EXEC PARS.PAS
...
SOURCE     : TEST.
LISTING    : TEST.LST
Option (n,p,u or <cr>): P
```

The parser program options are described further in Section D.2.

D.4 Generation of Tables for Fortran Parser

The tables PARS.TAB are transformed into block data statements by the program XFGEN. Option M can be used to generate block data statements suitable for 16, 32, 36, or 60 bit machines. The default option for M is 36. Tokens of the language are stored in an indexed-sequential structure. Reserved words can be up to 10 characters long. The file containing the block data statements is obtained by:

```
RUN XFGEN
TABLES  = PARS.TAB/M:36
OUTPUT  = PARS.OUT
BLKDAT  = BLKDAT.FOR
```

There are two Fortran parser template files available: XPRS.FOR and XPRSE.FOR. The second template file contains the routines to do error recovery when error productions are used. The following sequence of commands obtains an executable parser. A program to be parsed is assumed to be on file TEST.

```
EXE BLKDAT.FOR,XPRS.FOR
INPUT FILE  [PRS.INP] = TEST.
OUTPUT FILE [PRS.LST] = TEST.LST
SEMACT FILE [PRS.SEM] = TEST.SEM
;  A listing is produced on file TEST.LST,
;  the semantic actions called are written to file
;  TEST.SEM.
```

D.5 Simula Skeleton Compiler for DEC-10

The tables PARS.TAB can be transformed into a Simula class or into a text file. With the Simula class the tables are built into the compiler when the class is translated. With the text file the arrays of the skeleton compiler are initialized at run-time when the file is read. The lexical scanner tables are organized in an indexed-sequential manner. The transformations are carried out by:

```
RUN GENSIM
TABLES  = PARS.TAB
DO YOU WANT A SIMULA CLASS (Y OR N)?
; The simula class is written to file PARSTB.SIM,
; otherwise a text file PARS.SIM is generated.
```

The skeleton compiler consists of the files PARSER.SIM, DRIVER.SIM, and MAIN.SIM. The parser part of the skeleton compiler is a self-contained Simula class. MAIN.SIM is used when the tables are represented by a Simula class, DRIVER.SIM is used when the tables are on a text file.

D.6 Lisp Skeleton Compiler for DEC-10 and DEC-20

The skeleton compiler is written in UCI-Lisp. Arrays are used to store the parse tables. The transformation program produces two files from PARS.TAB: one file contains array allocations (for binary program space), the other file contains the parse tables in a form readable by the parser program. In this step error numbers together with their expected symbols can be inserted into an error message file. The transformation program prompts for required files and reports the amount of binary program space needed by the parser program.

```
RUN LSPGEN
...
; By default the file LSPTAB contains the parser tables,
; the file ALLC.LSP contains the array definitions.
```

The skeleton compiler is on file XPRS.LSP. The array allocation file must be loaded first when the system is built. The arrays are initialized by reading the parse table file. A file with semantic actions (see section 6.2) may be supplied also.

D.7 Generation of Tables in C

The tables on file PARS.TAB can be transformed into an initialization part in C. These tables (in the example on file TAB.C) together with the template parser XPRS.C can be used on a PDP-11.

```
RUN CGEN
TABLES      : PARS.TAB
TABC        : TAB.C
OUTPUT      : PARS.LST
```

D.8 Generation of Tables for an 8080

Grammars of the size of Pascal or C are suitable to generate parse tables for 8-bit machines. The second part of the table generator should be used with the options S and M to obtain tables which require the least amount of space.

The table transformation program first generates tables in a byte-oriented form. If requested the tables are further transformed into 8080 assembly data statements. The tables can be inserted into a skeleton compiler written in 8080 assembly language. The skeleton compiler is on file XPRS.808.

```
RUN XGEN80
TABLES  = PARS.TAB
...
; File SIM contains the tables in machine independent form,
; file PARS.808 contains the tables in 8080 assembly source.
```

The files PARS.808 and XPRS.808 can be moved to an 8080 micro-computer. On the DEC-10 these files can be translated with the 8080 assembler MAC80 and then interpreted with the 8080 interpreter INT80.

Appendix E

E.1 Parse Table Generation on CDC Cyber

The two parts of the XBSW system must be executed in sequence. It is assumed that the grammar is on file LANGRM. On a 64 character set machine character 0 (colon) may not be used to formulate the grammar input. The character 51 (percent) should be used instead. The resulting file PARSTAB, however, can be edited so that it may contain colons. The control card sequence to obtain the file PARSTAB is as follows:

```
GET,XBSW,XTRANS.
XBSW,LANGRM,OUTPUT,TEMP.
XTRANS,TEMP,OUTPUT,PARSTAB/S+.
```

The form in which the options on the control card for XTRANS are provided corresponds to the CDC-Pascal3 conventions. The available options are S, M, and L (see appendix A).

E.2 Pascal Skeleton Compiler for CDC

The tables obtained with the table generator are transformed with the program XPGEN into Pascal3 value parts, or with the program XXGEN into Pascal procedures. (Each procedure consists of a sequence of assignment statements).

The Pascal skeleton compiler is in UPDATE form on file XPRSPL. A source form of the skeleton compiler is used for inserting the tables (as value part or by procedures). If the tables are inserted as value part then the modification deck NPVMOD must be applied first to XPRSPL. The following control card sequence is suggested to obtain an executable parser.

```
GET,XPGEN,NPVMOD,XPRSPL.
UPDATE,P=XPRSPL,I=NPVMOD,L=0,C=PRSTMP,F,D,8.
XPGEN,PARSTAB,PRSTMP,PARSER,OUTPUT.
GET,PASCAL,PASCLIB.
PASCAL,PARSER.                        compile parser
LGO,<input>,<output>/<options>.    execute parser
```

The parser program may be used with options. The options are:

- N No program listing is produced.
- U An input line is 80 characters long instead of 72.
- P The semantic action number is recorded when the semantic routine is called.

E.3 Fortran Skeleton Compiler for CDC

The tables PARSTAB are transformed into block data statements. Option M can be used to generate block data statements suitable for 16, 32, 36, or 60 bit machines. The default option for M is 60. Tokens of the language are stored in an indexed-sequential structure (see section 5). Reserved words can be up to 10 characters long. The block data file is obtained by:

```
GET,XFGEN.
XFGEN,PARSTAB,OUTPUT,BLKDATA/M60
```

The Fortran skeleton compiler is in UPDATE form on file FPRSPL. The following control card sequence is suggested to obtain an executable parser:

```
REWIND,BLKDATA,LGO.
GET,FPRSPL.
FTN,I=BLKDATA.                must be first on LGO
UPDATE,P=FPRSPL,L=0,F.
FTN,I.                        compile parser
LGO,<input>,<output>.         execute parser
```

If error recovery by error productions is used then the modification deck ERRMOD must be applied first to the file FPRSPL.

Appendix F

The control cards described here are general suggestions on how to use the XBSW system with OS/VS and VM/370-CMS operating systems. For general use JCL procedures or command language procedures should be installed. Local documentation describing the differences in control cards using the Pascal system should be consulted.

F.1 Parse Table Generation on IBM-370

The two parts of the XBSW system must be executed in sequence. It is assumed that the grammar is on file LAN.GRM, and that the XBSW system is on file XBSW.LOAD. The following JCL is suggested to obtain the table file &PRSTAB.

```
// EXEC PGM=XBSW
//STEPLIB DD DSN=XBSW.LOAD,DISP=SHR
//LOCAL DD UNIT=SYSDA,SPACE=(CYL,10)
//SYSPRINT DD SYSOUT=A
//GRAMMAR DD DSN=LAN.GRM,DISP=(OLD,PASS)
//TMPTAB DD DSN=&TMP,DISP=(NEW,PASS),UNIT=SYSDA,
//    DCB=(RECFM=FB,LRECL=80,BLKSIZE=4000),SPACE=(TRK,(5,5))
// EXEC PGM=XTRANS
//STEPLIB DD DSN=XBSW.LOAD,DISP=SHR
//TMPTAB DD DSN=&TMP,DISP=(OLD,PASS)
//SYSPRINT DD SYSOUT=A
//TABLES DD DSN=&PRSTAB,DISP=(NEW,PASS),UNIT=SYSDA,
//    DCB=(RECFM=FB,LRECL=80,BLKSIZE=4000),SPACE=(TRK,(5,5))
```

The program XTRANS may be used with the options S, M, and L (see Appendix A) if the Pascal system in use allows for options to be passed to a program. Parameters would be defined as follows:

```
PARM='S  M'
```

In CMS the following command sequence is suggested. The file LAN GRAMMAR contains the grammar to be processed.

```
FILEDEF GRAMMAR DISK LAN GRAMMAR
FILEDEF SYSPRINT DISK LAN1 LISTING
RUN XBSW

FILEDEF SYSPRINT DISK LAN2 LISTING
FILEDEF TABLES DISK LAN TABLES
RUN XTRANS (S M
```

The output of the table generator is on the files LAN1 LISTING and LAN2 LISTING. The generated tables are on file LAN TABLES.

F.2 Fortran Skeleton Compiler

The tables on file &PRSTAB are transformed into Fortran block data statements. Tokens of the language are stored in an indexed-sequential structure (see section 5). Reserved words can be up to 16 characters long. The following JCL is suggested to obtain the block data file:

```
// EXEC PGM=GENFOR
//STEPLIB DD DSN=XBSW.LOAD,DISP=SHR
//TABLES DD DSN=&PRSTAB,DISP=(OLD,PASS)
//SYSPRINT DD SYSOUT=A
//BLKDAT DD DSN=&BLKDAT,DISP=(NEW,PASS),UNIT=SYSDA,
//    DCB=(RECFM=FB,LRECL=80,BLKSIZE=4000),SPACE=(TRK,(5,5))
```

It is assumed that the Fortran skeleton compiler is on file FORPRS. The block data statements and the skeleton compiler can now be compiled by Fortran (G or H). The semantic actions are recorded on unit 4 (FT04F001).

```
// EXEC ...  (Compile and Go version of Fortran)
//FORT.SYSIN DD DSN=&BLKDAT,DISP=(OLD,PASS)
//            DD DSN=FORPRS,DISP=(OLD,PASS)
//GO.SYSIN DD *
 ...
//GO.FT04F001 DD SYSOUT=A
```

Block data statements for 16, 36, or 60 bit machines can be generated by using the M option. Example:

```
PARM='M 36'
```

The default value for M is 32.

The CMS control command sequence to obtain a file containing the block data statements is as follows:

```
FILEDEF TABLES DISK LAN TABLES
FILEDEF SYSPRINT DISK LAN3 LISTING
FILEDEF BLKDAT DISK LANBLK FORTRAN
RUN GENFOR (M 32
```

The file LANBLK FORTRAN contains the block data statements. Under CMS the Fortran skeleton compiler is available as XPRS FORTRAN. The skeleton compiler containing the routines for error recovery by error productions is on file XPRSE FORTRAN.

F.3 Pascal Skeleton Compiler

Depending on the Pascal system available, the tables on file &PRSTAB are transformed with the program GENPAS either into Pascal procedures

or into a Pascal value part. (A Pascal procedure consists of a sequence of assignment statements). The procedures or the value part are inserted into a skeleton parser. It is assumed that the skeleton parser in Pascal is on file PASPRS. The tokens of the language are stored in an indexed-sequential structure. Reserved words can be up to 16 characters long. The following JCL is suggested to transform the parser tables:

```
// EXEC PGM=GENPAS
//STEPLIB DD DSN=XBSW.LOAD,DISP=SHR
//TABLES DD DSN=&PRSTAB,DISP=(OLD,PASS)
//SYSPRINT DD SYSOUT=A
//PARSIN DD DSN=PASPRS,DISP=(OLD,PASS)
//PARSOUT DD DSN=&PARSER,DISP=(NEW,PASS),UNIT=SYSDA,
//    DCB=(RECFM=FB,LRECL=80,BLKSIZE=4000),SPACE=(TRK,(5,5))
```

The resulting program on file &PARSER can be compiled with the Pascal compiler. The semantic actions are recorded on file SMFILE.

```
// EXEC ...  (Compile and Go version of Pascal)
//PAS.SYSIN DD DSN=&PARSER,DISP=(OLD,PASS)
//GO.SOURCE DD *
 ...
//GO.SYSPRINT DD SYSOUT=A
//GO.SMFILE DD SYSOUT=A
```

The CMS control command sequence for transforming the tables on file LAN TABLES into a Pascal value part and for its insertion into the skeleton parser XPRS PASCAL is as follows:

```
FILEDEF TABLES DISK LAN TABLES
FILEDEF PARSIN DISK XPRS PASCAL
FILEDEF PARSOUT DISK PARSER PASCAL
FILEDEF SYSPRINT DISK LAN4 LISTING
RUN GENPAS
```

The resulting program PARSER PASCAL can now be compiled with the Pascal compiler and executed.

Appendix G

The control commands are described here at the basic command level; for general use of the XBSW system the installation of command language procedures is suggested.

G.1 Parse Table Generation on VAX

The two parts of the XBSW system must be executed in sequence. It is assumed that the grammar to be processed is on file LAN.GRM.

```
ASSIGN LAN.GRM GRAMMAR          !grammar input
ASSIGN LAN1.LST PAS$OUTPUT      !listing
RUN XBSW

ASSIGN LAN2.LST PAS$OUTPUT      !listing
ASSIGN PARS.TAB TABLES          !generated tables
RUN XTRANS
```

The program XTRANS can be also used with the options S, M, and L (see Appendix A). For this purpose first a symbolic name must be defined so that options can be provided on the command line. Assuming that the system resides in directory [XXX] the following command lines will execute XTRANS with the options S and M:

```
XTRANS:==$[XXX]XTRANS.EXE
XTRANS /S,M
```

G.2 Pascal Skeleton Compiler for VAX/VMS

The tables obtained with the table generator are transformed with the program GENPAS for the VAX Pascal skeleton compiler. The tables are inserted as value part.

```
ASSIGN PARS.TAB TABLES          !tables from generator
ASSIGN XPRS.PAS PARSIN          !skeleton parser
ASSIGN PARSER.PAS PARSOUT       !generated parser
ASSIGN LAN3.LST PAS$OUTPUT      !listing
RUN GENPAS
```

The generated parser program can now be translated with the Pascal/VMS compiler and executed. The following example assumes that a program TEST is available.

```
PASCAL PARSER.PAS               !compile parser
LINK PARSER                     !create executable image

ASSIGN TEST SOURCE              !parser input
RUN PARSER
Enter options (N,P,U or <cr>):
```

The parser program options are described further in Section D.2.

G.3 Generation of Tables for VAX Fortran

The tables on file PARS.TAB are transformed into block data statements with the program GENFOR. Option M can be used to generate block data statements suitable for 16, 32, 36, or 60 bit machines. The default option for M is 32. Reserved words can be up to 16 characters long. If the M option is used first a symbolic name must be defined. The following example assumes that the program resides in directory [XXX].

```
ASSIGN PARS.TAB TABLES          !tables from generator
ASSIGN PARS.FOR BLKDAT          !block data statement file
ASSIGN LAN4.LST PAS$OUTPUT      !listing
GENFOR:==$[XXX]GENFOR.EXE
GENFOR /M32
```

There are two Fortran parser template files available: XPRS.FOR and XPRSE.FOR. The second template file contains the routines for error recovery when error productions are used. The following commands obtain an executable parser. A test program is assumed to be on file TEST.

```
FORTRAN PARS.FOR,XPRS.FOR       !compile parser
LINK PARS,XPRS

ASSIGN TEST INPUT               !parser input
ASSIGN TT OUTPUT                !listing on tty
ASSIGN SEM.LST SEM              !listing of semantic calls
RUN PARS
```

G.4 Generation of Tables in C

The tables on file PARS.TAB are transformed with the program GENC into an initialization part in C.

```
ASSIGN PARS.TAB TABLES          !tables from generator
ASSIGN TAB.C TABC               !initialization part in C
RUN GENC
```

The generated initialization part can be used together with the template parser XPRS.C and compiled with the C compiler.

G.5 Generation of Tables for an 8080

Grammars of the size of Pascal or C are suitable to generate parse tables for 8-bit machines. The program XTRANS should be used with the options S and M to obtain tables which require the least amount of space.

The tables PARS.TAB can be transformed into 8080 assembly statements with the program GEN8080.

```
ASSIGN PARS.TAB TABLES          !tables from generator
ASSIGN T8080.LST PAS$OUTPUT
ASSIGN PARS.808 ASSEM           !8080 assembly statements
RUN GEN8080
```

The files PARS.808 and XPRS.808 can be moved to an 8080 micro-computer and compiled there. The file T8080.LST contains a readable form of the 8-bit tables.