

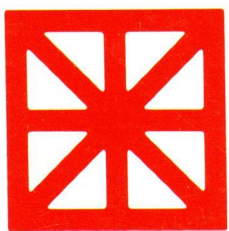
CR80 AMOS, PARSING SYSTEM  
USERS MANUAL

CSS/210/USM/0051

# CR80 minicomputer



Software




CHRISTIAN ROVSING A/S  
Copenhagen . Denmark



TITLE: CR80 AMOS, PARSING SYSTEM  
USERS MANUAL

DOCUMENT NO: CSS/210/USM/0051

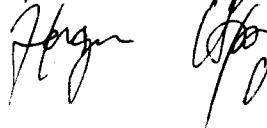
PREPARED BY: Lars Otto Kjær Nielsen



APPROVED BY: Jørgen Høg



AUTHORIZED BY: Jørgen Høg



DISTRIBUTION:

ISSUE:	1							
DATE:	800620							

LIST OF CONTENTS	PAGE
1 SCOPE.....	2
2 APPLICABLE DOCUMENTS.....	3
3 OVERVIEW.....	4
4 PARSE TABLE GENERATOR.....	7
4.1 Program activation syntax.....	7
4.2 Input file syntax (meta syntax).....	9
4.3 Object file format.....	12
4.4 Messages from the parse table generator.....	13
5 PARSING.....	17
5.1 Parsing scheme.....	17
5.1.1 Predefined symbols.....	22
5.1.2 Error recovery.....	22
5.2 SWELL parser.....	23
5.2.1 Interface description.....	24
5.2.2 Integration.....	26
5.3 PASCAL parser.....	28
5.3.1 Interface description.....	28
5.3.2 Integration.....	31
6 EXAMPLE.....	32

## 1. SCOPE.

This manual describes a parsing system, including three different tools:

- parse table generator (program)
- SWELL parser (link module)
- PASCAL parser (source texts)

The parse table generator is to be executed on a CR80 minicomputer running the AMOS operating system.

The parsers are to be integrated with programs defining the external interfaces, and they are not system dependant.

2. APPLICABLE DOCUMENTS.

- [1] CR80 PASCAL, REFERENCE MANUAL  
CSS/460/RFM/0001
- [2] SWELL 80, REFERENCE MANUAL  
CSS/415/RFM/0002
- [3] CR80 AMOS, TERMINAL OPERATING SYSTEM, USERS MANUAL  
CSS/380/USM/0026
- [4] CR80 AMOS, COMMAND INTERPRETER, USERS MANUAL  
CSS/381/USM/0037
- [5] Simple LR(k) Grammars, F.L. de Remer,  
Communications of the ACM, july 1971.
- [6] LR Parsing, A.V. Aho and S.C. Johnson,  
Computing Surveys, june 1974.

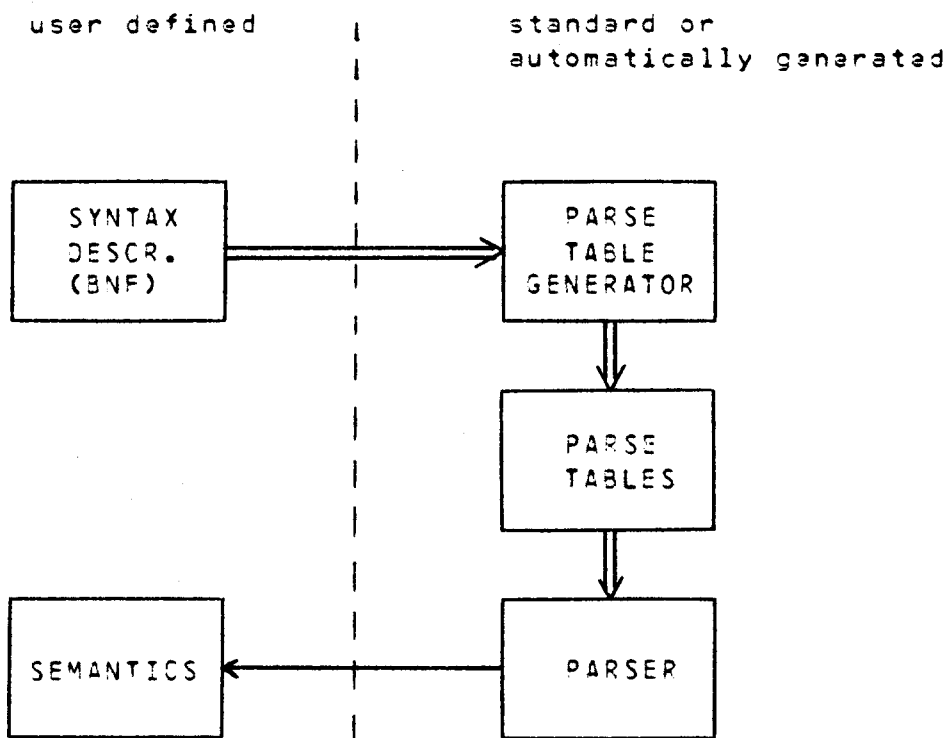
### 3. OVERVIEW.

The main purpose of the present parsing system is to bridge the gap between program input (on text level) and the actions performed by the actual program.

The main elements of the present parsing system are:

- table driven scanning of input text (recognizing input symbols like constants, identifiers etc.)
- table driven syntax analysis
- automatic generation of scan tables and syntax tables

The use of the parsing system may be visualized like this:



=> data flow  
-> control flow

When using the parsing system, the user must proceed according to the following scheme:

1. Create a syntax description file, defining the actual input language, using a modified Backus-Naur Form.
2. Generate a table file, holding scan table and syntax table for the actual language. This generation is performed by running the parse table generator program,

using the syntax description file as input file.

3. Write a semantics procedure to be activated by the parser when recognizable constructs are parsed. The semantics procedure should perform proper actions corresponding to the action index handed over by the parser.
4. Integrate parse tables, parser and semantics procedure, creating an entire program.



#### 4. PARSE TABLE GENERATOR.

The parse table generator reads a syntax description, checks it, generates parse tables and optionally prints information like production lists, parse states etc.

##### 4.1 Program activation syntax.

The syntax for calling the parse table generator is:

```

PARSERGEN { <option> }
                                     6
                                     3

<OPTION> ::= I:<file-id>
           ! P:<file-id>
           ! O:<file-id>
           ! F:<object-format>
           ! L:<list-mode>
           ! V:<verification-mode>

<object-format> ::= ABS ! REL ! OLD

<list-mode> ::= BNF ! ALL

<verification-mode> ::= YES ! NO

```

I:<file-id> selects the input file, from which the actual syntax is read (the meta syntax by which the input syntax is defined may be found in section 4.2). The "I" parameter must be present.

P:<file-id> selects the print file, on which input verification, production list, parse states and actions,

error messages and statistics are printed. The "P" parameter is optional and the current output file is used as default.

**O:**<file=id> selects the object file on which the parse tables are written. The "O" parameter must be present.

**F:**<object=format> defines the format of the parse tables to be generated. The primary difference is the way of pointing within the tables. ABS is used when generating tables for the PASCAL parser (using absolute indices within tables) and REL is used when generating tables for the SWELL parser (using relative word distances when addressing within parse tables). OLD is included only to be able to generate tables for the SWELL compiler, using an older version of the SWELL parser.

**L:**<list=mode> selects the amount of information printed by the parse table generator. In the figure below are indicated the subjects printed at different list modes:

subject to be printed	list mode		
	'default'	BNF	ALL
parse actions	-	-	+
parse states	-	-	+
productions	-	+	+
symbols	-	+	+
error messages	+	+	+
statistics	+	+	+

**V:**<verification=mode> defines whether the input text is to be listed. This may be relevant to pinpoint errors in the input syntax, as these errors are reported immediately when they are discovered. The "V" parameter may be omitted, and the default value is NO.

#### 4.2 Input file syntax.

The input file defining the syntax of a language is written according to a special meta syntax. To avoid conflicts between the meta syntax and the actual syntax to be processed, the symbols of the meta syntax (the meta symbols) are defined by the user.

Each meta symbol consists of exactly one character in the range 33..127.

Symbols used in the input syntax may consist of an arbitrary number of characters in the range 33..127 of which the first 16 characters are relevant. Neither nonterminal symbols nor any other symbols have to be surrounded by any kind of brackets.

In the input syntax any two symbols must be separated by one or more separator characters (in the range 1..32).

The input file syntax is shown below:

```
<input-syntax> ::= <meta-symbols>
                  <terminal-symbols>
                  <production-list>
                  "stop"

<meta-symbols> ::= "new" "alternative" "stop"

<terminal-symbols> ::= <symbol-list> "stop"

<symbol-list> ::= <symbol>
                  ! <symbol-list> <symbol>

<production-list> ::= <production>
                     ! <production-list> <production>

<production> ::= <new-production>
                ! <production> <alternative-production>

<new-production> ::= "new" <symbol> <right-hand-part>

<alternative-production> ::= "alternative" <right-hand-part>

<right-hand-part> ::= <symbol-list>
                    ! <empty>
```

Another notation of the same syntax is:

```

"new" "alternative" "stop"

{ <symbol> } * "stop"
  4

{ "new" <symbol> { <symbol> } *
  0

{ "alternative" { <symbol> } * } * } *
  0 0 0

"stop"

```

A number of conventions are embedded in the system. These conventions are listed and commented below:

- The first four terminal symbols in the input syntax must be synonyms for:
  - identifier
  - constant
  - string
  - error-symbol
 and they must all be listed, even when the actual syntax does not use all of them. They must be present because the scanning part of the parser knows these symbols on beforehand (and uses the lower symbol numbers for them). The user must define synonyms for them, because predefined names might conflict with user defined symbols of a syntax.
- Terminal symbols must be defined in accordance with the character classes used by the parser. When disregarding

the first four (standard) terminal symbols, a terminal symbol must consist of alphanumeric only (with a leading alpha) or it may consist of delimiter symbols only.

- The right hand part of a production may be empty. However, the SLR1 scheme used by the parse table generator will only be able to handle a single or a very small number of productions having an empty right hand part.
- The right hand part of a production may not contain more than 15 symbols.

#### 4.3 Object file format.

When activating the parse table generator, the user may specify the object file format by using the "F" option.

The object files differ in their logical contents (using RELative pointing for the SWELL parser and ABSolute pointing for the PASCAL parser). However, the object files also differ in their representation.

RELative object files (for the SWELL parser) are text files, containing the tables as one list of hexadecimal constants separated by commas. This file is suitable for merging into the INIT part of a SWELL module.

ABSolute object files (for the PASCAL parser) are binary files containing the tables as a string of 16 bits integers to be read by the initiating part of the parser at run time.

#### 4.4 Messages from the parse table generator

In the following are listed the (error) messages, that may be generated by the parse table generator.

##### ACTION TABLE FULL

The (internal) action table within the parse table generator has been overfilled. This table is controlled by the constant MAXACTIONS.

##### CONNECT CURR OUT FILE <cc>

The completion code <cc> was returned, when the program tried to connect a stream to the current output file.

##### CONNECT INPUT FILE <cc>

##### CONNECT PARAM FILE <cc>

##### CONNECT PRINT FILE <cc>

##### CONNECT OBJECT FILE <cc>

As for connect curr out file.

##### FIND FILE <cc>

One of the files in the parameter list could not be found.

##### HASH TABLE OVERFILLED

The internal hash table used for symbols of the BNF has been overfilled. The hash table is controlled by the constant MAXSYMBOL.

##### ILLEGAL FORMAT SPECIFICATION

The mneumo indicating object file format at the "F" parameter was illegal.

##### INELEMENT <cc>

The completion code <cc> was returned when reading from the parameter file.

## INFILEID &lt;cc&gt;

The completion code <cc> was returned when reading from the parameter file.

## INPUT FILE MISSING

The "I" parameter was missing in the parameter list.

## INPUT STREAM ERROR &lt;cc&gt;

The completion code <cc> was returned when reading from the input file.

## ITEM TABLE FULL

An internal table within the parse table generator has been overfilled. The table is controlled by the constant MAXITEMS.

## LEFT HAND SYMBOL MISSING

After the meta symbol <new> (for 'new production') a symbol is expected (the left hand symbol of a production). This symbol was not present.

## METASYMBOL ILLEGAL

The only metasyMBOL allowed after the list of terminal symbols is the <stop> symbol.

## MULTIPLY DECLARED &lt;symbol&gt;

A symbol can only be defined explicitly as a left hand symbol a single time. A number of productions using the same left hand symbol must be written as one 'new production' followed by a number of 'alternative productions'.

## OBJECT FILE MISSING

The "O" parameter was not included in the parameter list.

## OBJECT FORMAT NOT DEFINED

The "F" parameter was not included in the parameter list.



## OBJECT STREAM OUTPUT &lt;cc&gt;

The completion code <cc> was returned when writing on the object file.

## PARAMETER SYNTAX

A syntax error has been detected at a parameter (missing ':' or the like).

## PARAM STREAM ERROR &lt;cc&gt;

The completion code <cc> was returned when reading from the parameter file.

## PRINT STREAM ERROR &lt;cc&gt;

The completion code <cc> was returned when writing on the print file.

## PRODUCTION LIST FULL

The internal production list table within the parse table generator program has been overfilled. This table is controlled by the constant MAXPRODLIST.

## REDUCE/REDUCE CONFLICT ON &lt;symbol&gt;

The input syntax was not a proper SLR1 syntax. A parse state contains the ambiguity, that two possible productions are not distinguishable by looking at a single follower symbol. However, the parse tables will be generated with the convention that the first production is used. Grammars with that kind of problems should be used with great care.

## SHIFT/REDUCE CONFLICT ON &lt;symbol&gt;

The input syntax was not a proper SLR1 syntax. A parse state contains the ambiguity, that one production and the first part of another production are not distinguishable by looking at a single follower symbol. However, the parse tables will be generated, using the convention that the symbols are expected to be the first part of the longer of the two productions.

SYMBOLS NOT ACCESSIBLE FROM <GOAL-SYMBOL>: <symbol-list>

It is checked by the parse table generator, that all symbols of the input syntax (except for the first four standard symbols) are really used in the language.

SYMBOL TABLE FULL

The symbol table of the parse table generator has been overfilled. This table is controlled by the constant MAXSYMBOL.

SYNTAX ERROR

A syntax error has been discovered within the parameter list (the attribute of a parameter has been of wrong kind or the like).

UNDECLARED: <symbol>

A symbol has been used in the right hand part of a production without being included in the list of terminal symbols and without being defined as the left hand symbol of a production.

## 5. PARSING.

In this chapter is described the environment in which the parsing is performed. The environment is described functionally in section 5.1 and the actual interfaces to the SWELL parser and the PASCAL parser are defined in section 5.2 and 5.3.

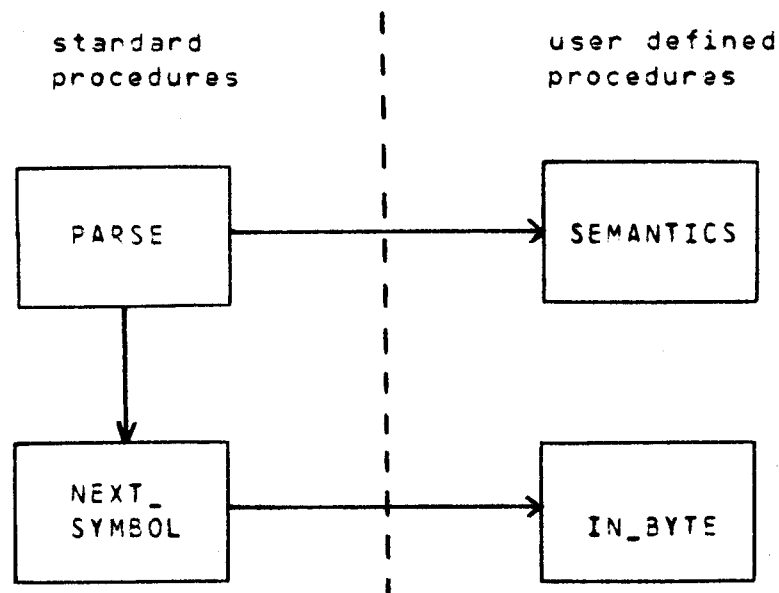
The principles of LR parsing are described in [5] and [6].

### 5.1 Parsing scheme.

The main objects used during parsing are the parse tables and a parse stack. The parse tables are not relevant to the user. However, the processing on the parse stack is reflected in the activation of the semantics procedures.

The parse stack primarily contains information describing the parse state. The total parse state is defined by a state value associated to each symbol in the parse stack. Similarly the user usually describes the semantic contents of each symbol on the parse stack in a set of associated values. The exact implementation of the semantics descriptions are described in section 5.2 and 5.3.

The control flow in a parsing system may be visualized like this:



PARSE is the controlling part of the parsing system, calling NEXT\_SYMBOL when the next symbol of the source is to be analyzed syntactically, and calling the SEMANTICS procedure in each of the following cases:

- A terminal symbol having attributes, has been met and is to be pushed on the parse stack (identifier, constant or string).
- A syntactical error has been detected.
- A production of the input language has been recognized.
- An escape character has been met in the source.

The scanner procedure NEXT\_SYMBOL activates the user defined procedure IN\_BYTE whenever the next input character is to be scanned. This is so to enable the user to implement special features on the source (error handling, file merging etc.) and to make the parse procedures system independent.

When the SEMANTICS procedure is activated, the parser delivers an action index, indicating which syntactical construction has been analyzed or what kind of error has occurred.

The semantic actions are listed below with the action index and a description of the accompanying information:

- 0: An ESCAPE character has been met during scanning. The parse stack pointer defines the top of the parse stack.
- 1: An IDENTIFIER has been met. The identifier is stored in the character array: SYMBOLBUF. The parse stack pointer points at the entry into which the user may put relevant information about the identifier.
- 2: A CONSTANT has been met. The value of the (integer) constant is delivered in the ATTRIBUTE parameter and the parse stack pointer points at the entry into which the user may put relevant information about the constant (e.g. the value).
- 3: A STRING has been met. The string is stored in the character array: SYMBOLBUF. The parse stack pointer points at the entry into which the user may put relevant information about the string. The ATTRIBUTE parameter delivers the size of the string in characters.
- 4: A SYNTAX ERROR has been discovered. The ATTRIBUTE parameter indicates the kind of error. The user is

responsible for reporting the error, while the parser tries to recover from the error automatically. The possibilities for the parser to recover successfully depend on the existence of the error-symbol in the user defined syntax. During recovery the parser may call the semantics procedure with the error action a number of times, and the user may have to suppress superfluous error reporting.

The error codes delivered id the ATTRIBUTE parameter are:

- 1 constant overflow
- 2 unexpected symbol (syntax error)
- 3 string syntax
- 4 string size
- 5 parse stack full
- 6 recovery failed

>= 5: A PRODUCTION of the input syntax has been recognized. The action index corresponds to the production number in the production list generated by the parse table generator. The parse stack pointer points at the first one of the stack entries describing right hand symbols of the production. This entry will be used to describe the left hand symbol of the production after the reduction.

If the action index corresponds to the production:

TERM ::= ( EXPRESSION )

then the parse stack pointer points at the stack entry corresponding to "(". At exit from the semantics procedure this entry should describe the left hand symbol "TERM", and it is most likely, that the primary action of the semantics procedure in this case will be to move the contents of the "EXPRESSION" entry to the "(" entry, creating a valid "TERM" entry.

### 5.1.1 Predefined symbols.

During scanning the parser distinguishes among the following character classes:

- ALFA (letters)
- NUMERIC (digits)
- DELIMITER
- COMMENTCHAR
- STRINGCHAR
- HEXCHAR
- IGNORE
- ESCAPE

These classes are used when assembling and recognizing symbols of the input language.

A number of symbols and constructs are predefined within the parser. These symbols are described in the following.

IDENTIFIERS are build up by ALFAs and NUMERICs. The first character of an IDENTIFIER must be an ALFA.

CONSTANT may be a decimal constant or a hexadecimal constant. A decimal constant contains NUMERICs only. A hexadecimal constant is prefixed with a HEXCHAR and contains 1 to 4 digits or hex letters ('A' to 'F').

STRING is a simple string or a concatenated list of simple strings. A simple string is a series of characters surrounded by STRINGCHARs. Within a simple string, characters in the range 32..126 may be used directly. Special characters may be included by writing their character value as a decimal constant surrounded by the brackets '(' and ')'. Simple strings may be concatenated by using '&' as a catenation operator.

COMMENT is a series of characters prefixed with a COMMENTCHAR and terminated with a COMMENTCHAR or a line

terminating character (in the range 1..31).

### 5.1.2 Error recovery.

The parser includes facilities for performing error recovery. In case of a syntax error, the recovery procedure (within the parser) is activated, and it performs like this:

```
if specific "recover symbols" are included
  in the parse tables then
begin
  repeat read and skip next input symbol
  until a recover symbol is found;
  repeat skip the parse state
    at the top of the parse stack
  until a state is found, after which
    an "error symbol" is acceptable;
end;
```

As it appears from this algorithm, the recovery is based on the existence of some "recover symbols" and the "error symbol". If they are not present, the parsing will terminate with the cause, "unrecoverable".

It is up to the user to define these symbols. The "recover symbols" are automatically (by the parse table generator) generated as the set of terminal symbols, that may appear as a follower symbol after the "error symbol". Thus the user just has to include the "error symbol" at relevant points in the syntax. It is hard to give specific rules for the insertion of the "error symbol". However, an example may



illustrate the use of the error symbol:

In the grammar of PASCAL it would be reasonable to have the following production for statement:

statement ::= error-symbol statement

As statement may be empty in PASCAL, the symbols following error-symbol will then be all symbols that may be leading symbols of a statement (BEGIN, IF REPEAT etc.) and symbols that may follow a statement (END, ";", UNTIL etc.).

## 5.2 SWELL parser.

The parse stack used by the SWELL parser is a stack of records of equal size. The first field of each record is an integer field reserved for use by the parser. This convention is introduced to minimize the use of index registers (the register usage is described in section 5.2.1), as the parser and the user defined semantics procedure may share a single register when accessing a specific entry in the parse stack.

The SWELL parser is prepared for dynamic selection and reselection of parse tables, and thus it is possible to define an overall input language consisting of nested languages. When the PARSE procedure of the SWELL parser is activated, the context is saved (in a work area in the parse table) and the former parse table (if any) is chained to the actual parse table. At exit from the PARSE procedure (i.e. when the final production has been recognized) the actual parse table is unstacked, the context is reestablished and execution may proceed in an outer parse table. However, a parse table cannot be used recursively.

## 5.2.1 Interface description.

The interface to the SWELL parser consists of the following objects:

```
Procedures:
  PARSE          (standard procedure)
  IN_BYTE       (user written)
  <semantics>   (user written)
Variables:
  SYMBOLBUF     (standard array of char)
```

The declarations of these objects are shown and commented in the following:

```
PROCEDURE PARSE
(STACKATTRIBUTE SIZE,
 MAXSTACKATTRIBUTES: INTEGER;
 R4; "entry point of semantics procedure
 R5; "parse stack base address
 R7; "parse table base address
 R6); "link
```

STACKATTRIBUTE SIZE defines the size of each record in the parse stack (in words).

MAXSTACKATTRIBUTES defines the limit of the parse stack (the maximum number of records in the parse stack)

R4 defines the semantics procedure to be activated by the parser (may be set by: location("semantics"));

R5 defines the base address of the parse stack (may be set by: address("parsestack")).

R7 defines the base address of the parse tables (may be set by: address("parsetable")).

IN\_BYTE is a user written procedure, and it should be declared like this:

```
PROCEDURE IN_BYTE
(R3; "character value (return)
R6); "link
    "all other registers are unchanged
```

The semantics procedure is a user written procedure transferred to the parser as a parameter, so the name of the procedure is not relevant to the parser. The semantics procedure should be declared like this:

```
PROCEDURE SEMANTICS
(R1; "action number
R2; "attribute
R5; "parse stack pointer
R6); "link
    "all registers are unchanged at return
```

The symbol buffer holding the last scanned identifier or string is a standard array defined like this:

```
SYMBOLBUF: ARRAY [0..MAXSYMBOLLENGTH] OF CHAR;
```

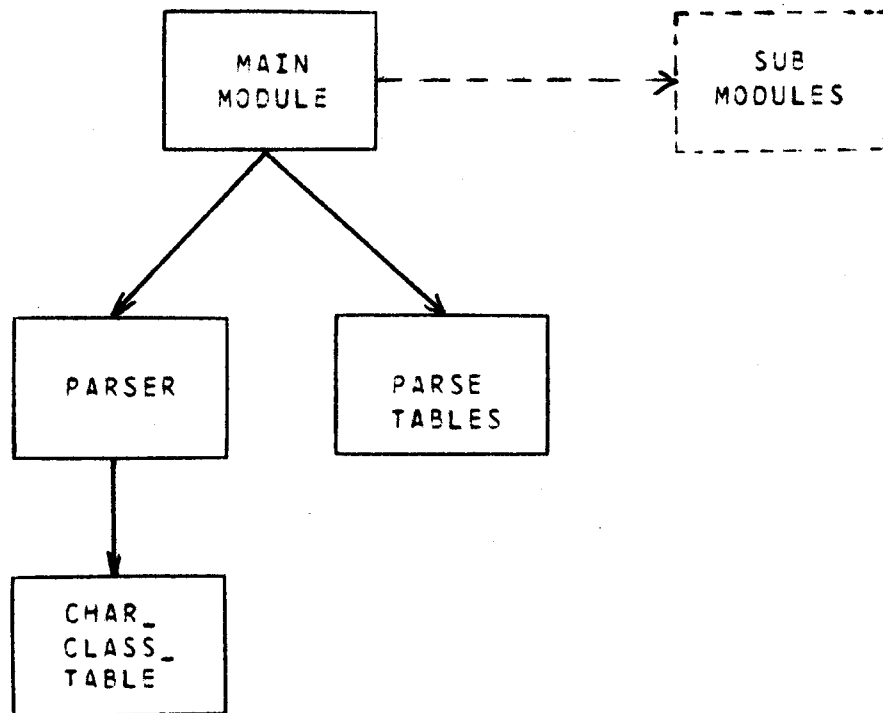
The standard value of MAXSYMBOLLENGTH is 132 characters.

### 5.2.2 Integration.

The SWELL parser is a link (sub)module to be linked together with a user defined main module and possibly some additional modules.

The parser imports the procedure IN\_BYTE and a character class table CHARCLASSTABLE. A standard character class table is offered as a link module in the parsing system.

It may often be convenient for the user to create the parse tables in an individual link module, and the parsing program is then structured like this:



The standard files offered for SWELL parsing are all found in the directory:

@\*\*GENS.D\*PARSER.D

The files are:

PARSER.L	std parser link module
PARSER.I	std parser import source
CHARCLASSTABLE.L	std character class table link module
CHARCLASSTABLE.I	std character class table import source

### 5.3 PASCAL parser.

The parse stack used by the PASCAL parser only contains parsing information. The user should then create a parallel stack of records to hold the semantic attributes of the symbols on the stack (an array of records having tags corresponding to nonterminal symbols of the grammar will often do). The PASCAL parser does not offer nested parsing with different languages, and it uses a fixed, predefined character classification for the scanning.

The parse tables for the PASCAL parser are loaded from the table file at run time. This is controlled by a standard 'INIT\_PARSE' procedure within the parsing system, using a user defined procedure, interfacing to the environment.

#### 5.3.1 Interface description.

The interfaces to the PASCAL parser consists of the following objects:

Procedures:  
INIT\_PARSE (standard procedure)  
NEXT\_TABLE\_WORD (user written procedure)  
PARSE (standard procedure)  
IN\_BYTE (user written procedure)  
SEMANTICS (user written procedure)

Variables:  
SYMBOLBUF (standard array of char)

Constants:  
MAXSYMBOLLENGTH (user defined size of SYMBOLBUF)  
MAXSTACK (user defined size of parse stack)  
MAXSCANENTRY (size of scan table)  
MAXPRODUCTION (size of production table)  
MAXACTION (size of action table)

INIT\_PARSE is called by the user to load and initialize parse tables and parse variables. The procedure has no parameters.

NEXT\_TABLE\_WORD is a user written procedure, called by the INIT\_PARSE procedure when reading the parse table file. The procedure must match the following declaration:

```
PROCEDURE NEXT_TABLE_WORD (VAR I: INTEGER);
```

PARSE is called by the user to start parsing. The procedure has no parameters.

IN\_BYTE is a user written procedure, called by the scanning part of the parser. The procedure must match the following declaration:

```
PROCEDURE IN_BYTE (VAR CH: CHAR);
```

SEMANTICS is a user written procedure performing the

semantic actions corresponding to syntactical constructs recognized by the parser. The procedure must obey the following declaration.

```
PROCEDURE SEMANTICS (ACTIONINDEX,  
                    ATTRIBUTE,  
                    STACKINDEX: INTEGER);
```

ACTIONINDEX selects the action to be performed (cf. section 5.1).

ATTRIBUTE delivers the value of a constant, the size of a string or an error code.

STACKINDEX is the stack pointer, used by the parser for indexing in the parse stack, and used by the user for indexing in the parallel stack of semantic attributes.

The character array holding the last scanned identifier or string is declared like this:

```
SYMBOLBUF: ARRAY [0..MAXSYMBOLLENGTH] OF CHAR;
```

The user must define the following set of constants:

MAXSYMBOLLENGTH defines the largest number of characters that the parser is capable of assembling as a single symbol.

MAXSTACK defines the size of the parse stack. This value depends very much on the input syntax. However, languages like SWELL or PASCAL would require less than 100 elements in the parse stack to compile usual programs.

MAXSCANENTRY, MAXPRODUCTION and MAXACTION should by the user be set to the values printed by the parse table generator (within the statistics).



The PASCAL parser uses a predefined character classification. This classification is defined like this:

ALFA =	'A'..'Z' and '_'
NUMERIC =	'0'..'9'
COMMENTCHAR =	"
STRINGCHAR =	'
HEXCHAR =	#
ESCAPE =	%
IGNORE =	characters in the range 1..32
DELIMITERS =	all others

### 5.3.2 Integration.

The PASCAL parser is a set of text files to be merged into a user written program. The files are all found in the directory:

@\*\*GENS.D\*PARSE.D

The files are:

CONSTS.S	to be merged into constant part
TYPES.S	to be merged into type part
VARS.S	to be merged into var part
PROCS.S	to be merged into procedure part

The procedures, variables and constants included in the user interface are named as indicated in section 5.3.1. All other names introduced by the parser files are prefixed by 'PRS\_' to avoid name conflicts.

## 6. EXAMPLE.

The example used to illustrate the use of the SWELL and the PASCAL parsers is a very limited calculator program primarily building on the expression syntax, known from almost all papers on parsing subjects.

Written in usual BNF, the grammar looks like this:

- ```

1 <goal-symbol> ::= <expression> =
2 <expression> ::= <term>
   ! <expression> + <term>
3 <term> ::= <factor>
   ! <term> * <factor>
4 <factor> ::= <constant>
   ! ( <expression> )

```

In the following is shown, how SWELL and PASCAL calculator programs may be generated when using the parsing system.

The syntax is written into a syntax description file, using the metasyntax as defined for the parse table generator:

```

" " #
NAME CONSTANT STRING ERROR
= + * ( )
" CALCULATION      EXPRESSION =
" EXPRESSION       TERM
"                  EXPRESSION + TERM
" TERM             FACTOR
"                  TERM * FACTOR
" FACTOR           CONSTANT
"                  ( EXPRESSION )

```

*start for  
calculator*

→

→

*start for  
calculator*

The parse table is generated by running the parse table generator program PARSEGEN. The object format is selected according to the destination language.

Overleaf is shown the printout generated by the parse table generator when activated with "L:ALL". The parse states and actions are usually of no relevance to the user. However, in case of syntax definitions causing reduce/reduce or shift/reduce conflicts the parse states and actions may illustrate the reasons for the conflicts.

In case a user wants to understand these states and actions fully, he may for instance study [5] and [6].

SYMBOLS:

|        |                |               |         |
|--------|----------------|---------------|---------|
| 1 NAME | 2 CONSTANT     | 3 STRING      | 4 ERROR |
| 5 =    | 6 +            | 7 *           | 8 (     |
| 9 )    | 10 CALCULATION | 11 EXPRESSION | 12 TERM |

PRODUCTIONS:

|    |               |     |                   |
|----|---------------|-----|-------------------|
| 5  | <GOAL SYMBOL> | --> | CALCULATION       |
| 6  | CALCULATION   | --> | EXPRESSION *      |
| 7  | EXPRESSION    | --> | TERM              |
| 8  |               | --> | EXPRESSION + TERM |
| 9  | TERM          | --> | FACTOR            |
| 10 |               | --> | TERM * FACTOR     |
| 11 | FACTOR        | --> | CONSTANT          |
| 12 |               | --> | ( EXPRESSION )    |

PARSE STATES AND ACTIONS:

|             |  |     |                     |
|-------------|--|-----|---------------------|
| #           |  | --> | # CALCULATION       |
| CALCULATION |  | --> | # EXPRESSION *      |
| EXPRESSION  |  | --> | # TERM              |
| EXPRESSION  |  | --> | # EXPRESSION + TERM |
| TERM        |  | --> | # FACTOR            |
| TERM        |  | --> | # TERM * FACTOR     |
| FACTOR      |  | --> | # CONSTANT          |
| FACTOR      |  | --> | # ( EXPRESSION )    |

|    |       |    |                |
|----|-------|----|----------------|
| 1: | SHIFT | 9  | ON CALCULATION |
| 2: | SHIFT | 9  | ON EXPRESSION  |
| 3: | SHIFT | 12 | ON TERM        |
| 4: | SHIFT | 14 | ON FACTOR      |
| 5: | SHIFT | 15 | ON CONSTANT    |
| 6: | SHIFT | 16 | ON (           |
| 7: | ERROR |    |                |

|   |  |     |               |
|---|--|-----|---------------|
| # |  | --> | CALCULATION # |
|---|--|-----|---------------|

|    |        |   |  |
|----|--------|---|--|
| 8: | REDUCE | 5 |  |
|----|--------|---|--|

|             |  |     |                     |
|-------------|--|-----|---------------------|
| CALCULATION |  | --> | EXPRESSION # *      |
| EXPRESSION  |  | --> | EXPRESSION # + TERM |

|     |       |    |      |
|-----|-------|----|------|
| 9:  | SHIFT | 18 | ON = |
| 10: | SHIFT | 19 | ON + |
| 11: | ERROR |    |      |

|            |  |     |                 |
|------------|--|-----|-----------------|
| EXPRESSION |  | --> | TERM #          |
| TERM       |  | --> | TERM # * FACTOR |

|     |        |    |      |
|-----|--------|----|------|
| 12: | SHIFT  | 21 | ON * |
| 13: | REDUCE | 7  |      |

|      |  |     |          |
|------|--|-----|----------|
| TERM |  | --> | FACTOR # |
|------|--|-----|----------|

|     |        |   |  |
|-----|--------|---|--|
| 14: | REDUCE | 9 |  |
|-----|--------|---|--|

|        |  |     |            |
|--------|--|-----|------------|
| FACTOR |  | --> | CONSTANT # |
|--------|--|-----|------------|

|     |        |    |  |
|-----|--------|----|--|
| 15: | REDUCE | 11 |  |
|-----|--------|----|--|

```

FACTOR          --> ( # EXPRESSION )
EXPRESSION      --> # TERM
EXPRESSION      --> # EXPRESSION + TERM
TERM            --> # FACTOR
TERM            --> # TERM + FACTOR
FACTOR          --> # CONSTANT
FACTOR          --> # ( EXPRESSION )

```

```

16:  SHIFT      23 ON EXPRESSION
17:  GO TO      3

```

```

CALCULATION     --> EXPRESSION = #

```

```

18:  REDUCE     6

```

```

EXPRESSION      --> EXPRESSION + # TERM
TERM            --> # FACTOR
TERM            --> # TERM + FACTOR
FACTOR          --> # CONSTANT
FACTOR          --> # ( EXPRESSION )

```

```

19:  SHIFT      26 ON TERM
20:  GO TO      4

```

```

TERM            --> TERM * # FACTOR
FACTOR          --> # CONSTANT
FACTOR          --> # ( EXPRESSION )

```

```

21:  SHIFT      28 ON FACTOR
22:  GO TO      5

```

```

FACTOR          --> ( EXPRESSION # )
EXPRESSION      --> EXPRESSION # + TERM

```

```

23:  SHIFT      19 ON +
24:  SHIFT      29 ON )
25:  ERROR

```

```

EXPRESSION      --> EXPRESSION + TERM #
TERM            --> TERM # * FACTOR

```

```

26:  SHIFT      21 ON *
27:  REDUCE     8

```

```

TERM            --> TERM * FACTOR #

```

```

28:  REDUCE     10

```

```

FACTOR          --> ( EXPRESSION ) #

```

```

29:  REDUCE     12

```

STATISTICS:

```

TERMINALS:      9
NONTERMINALS:   4
PROD.LIST SIZE: 24
ITEMS:          53
STATES:         15

```

```

SCANENTRIES.... 7
PRODUCTIONS.... 8
ACTIONS.....    31

```

TOTAL SIZE OF PARSE TABLES: 107 INTEGERS.

In case of a SWELL parsing program the rest of the generation is performed according to the following steps:

- The parse table may be converted to a link module by writing a submodule as shown below (using the parse table file DEMOBNF.H as a source file):

```
SUBMODULE DEMOBNF;  
EXPORT VAR CALCTABLE: ARRAY [0..107] OF INTEGER;  
INIT CALCTABLE =  
XSOURCE DEMOBNF.H  
D;  
ENDMODULE
```

- The main module, containing the semantics of the calculator, is shown overleaf:

MAINMODULE CALCULATOR;

CONST  
STACKMAX = 15;

%SOURCE @\*\*GENS.D\*SWELLPREFIX.D\*GENERALPARAMS.S  
%SOURCE @\*\*GENS.D\*SWELLPREFIX.D\*MONITORNAMES.S  
%SOURCE @\*\*GENS.D\*SWELLPREFIX.D\*IOSPARAMS.S  
%SOURCE @\*\*GENS.D\*UTILITYHELP.D\*.I  
%SOURCE @\*\*GENS.D\*PARSE.D\*PARSER.I

IMPORT VAR  
CALCTABLE: ARRAY [0..0] OF INTEGER;

TYPE  
ATTRIBUTE =  
RECORD  
PARSEINDEX: INTEGER;  
VALUE: INTEGER;  
END;

VAR  
SEM\_STACK: ARRAY [1..STACKMAX+SIZE(ATTRIBUTE)] OF INTEGER;

EXPORT PROCEDURE IN\_BYTE  
"\*\*\*\*\*"  
(R3; "CHARACTER VALUE (RETURN)  
R6); "LINK

VAR SAVER4, SAVER6: INTEGER;  
BEGIN  
R4=>SAVER4;  
R6=>SAVER6;  
INB(ADDRESS(CINFILETYPE)=>R4, R3, R6);  
SAVER4=>R4;  
EXIT(SAVER6);  
END;

PROCEDURE CALCULATE  
"\*\*\*\*\*"  
(R1; "ACTION NO  
R2; "ATTRIBUTE  
R5; "PARSE STACK POINTER  
R6); "LINK  
VAR SAVEREGS: ARRAY [0..7] OF INTEGER;

LONGWORK: LONG;  
WRK: ARRAY [0..7] OF INTEGER;  
BEGIN  
R7=>SAVEREGS[7];  
STC(6, ADDRESS(SAVEREGS[7])=>R7);

R5=>R6+SIZE(ATTRIBUTE);  
R5=>R7+(2\*SIZE(ATTRIBUTE));

CASE R1 OF

2: " CONSTANT  
R2=>R5@ATTRIBUTE.VALUE;

4: " ERROR  
MON(TERMINATE, R2=>R0, 0=>R1, R7);

6: " CALCULATION ::= EXPRESSION = "  
BEGIN  
(ADDRESS(COUTFILETYPE)=>R4)@FILETYPE.S=>R4;  
MON(STREAM, OUTINTEGER, ADDRESS(WRK)=>R0, R4,  
R5@ATTRIBUTE.VALUE=>R2, 0=>R3, R7): BIN\_EXIT;  
MON(STREAM, OUTNL, R4, R7): BIN\_EXIT;  
MON(STREAM, FLUSH, R4, R7): BIN\_EXIT;  
END;

8: " EXPRESSION ::= EXPRESSION + TERM "  
R5@ATTRIBUTE.VALUE+(R7@ATTRIBUTE.VALUE=>R0);

10: " TERM ::= TERM \* FACTOR "  
BEGIN  
ADDRESS(LONGWORK)=>R4;  
R5@ATTRIBUTE.VALUE=>R4@LONG.LEAST;  
R4@LONG+(R7@ATTRIBUTE.VALUE=>R0);  
R4@INTEGER=>R5@ATTRIBUTE.VALUE;  
END;

12: " FACTOR ::= ( EXPRESSION ) "  
R6@ATTRIBUTE.VALUE=>R0=>R5@ATTRIBUTE.VALUE;

END;

UNS(7, ADDRESS(SAVEREGS[0])=>R7);  
EXIT(R6);  
END;

BEGIN

```
ACCEPTFILES(R6);
READSYSPARAMS(R6);
OPENSTREAM(ADDRESS(CINFILETYPE)=>R4, INPUT_MODE=>R3, R6);
(ADDRESS(COUTFILETYPE)=>R4)@FILETYPE.S=>R4;
MON(STREAM, OUTTEXTB, R4, ADDRESS('READY(:10:)(:0:)' )=>R6, R7): BIN_EXIT;
MON(STREAM, FLUSH, R4, R7): PIN_EXIT;
PARSE(SIZE(ATTRIBUTE), STACKMAX, LOCATION(CALCULATE)=>R4, ADDRESS(SEM_STACK)=>R5,
      ADDRESS(CALCTABLE)=>R7, R6);
MON(TERMINATE, 0=>R0, 0=>R1, R7);
END;
ENDMODULE
```

- Finally the modules are linked together, creating the object program.

In case of a PASCAL program, it only remains to create the main program into which the standard parsing source files are merged at compile time. The PASCAL version of the calculator is shown overleaf:



```
%VOLIST
$D**GENS.D*PREFIX
%LIST
%EXECLEVEL=2
%WORKAREA=2000
%STREAMS=3
%FDS=6
%IOCBS=6
%TTLES=16
%MESSAGES=5
%VERSION=1
```

```
CONST
" CONFIGURATION "
MAXSCANENTRY = 7;
MAXPRODUCTION = 9;
MAXACTION = 31;
MAXSTACK = 25;
MAXSYMBOLLENGTH = 132;
```

```
$D**GENS.D*PARSE.D*CONSTS.S
$D**GENS.D*PARSE.D*TYPES.S
```

```
VAR
```

```
$D**GENS.D*PARSE.D*VARS.S
```

```
SEM_STACK: ARRAY [1..MAXSTACK] OF INTEGER;
I_STREAM, O_STREAM: STREAM;
FROM_ADAM: BOOLEAN;
FSN: FILE_SYSTEM_NAME;
VOL: VOLUME_NAME;
NAMELIST: NAMELISTTYPE;
NAMENO: INTEGER;
F, TABLEFILE: FILE;
CC: COMPLETION_CODE;
CH: CHAR;
```

```
PROCEDURE IN_BYTE (VAR CH: CHAR);
BEGIN
  INBYTE(I_STREAM, CH, CC);
  IF CC <> IO_OK THEN TERMINATE(CC);
END;
```

```
PROCEDURE NEXT_TABLE_WORD (VAR I: INTEGER);
BEGIN
  INWORD(I_STREAM, I, CC);
  IF CC <> IO_OK THEN TERMINATE(CC);
```

```
END;
```

```
PROCEDURE SEMANTICS (ACTIONNO, ATTRIBUTE, STACKINDEX: INTEGER);
```

```
BEGIN
```

```
CASE ACTIONNO OF
```

- 2: " STACK "
 SEM\_STACK[STACKINDEX]:= ATTRIBUTE;
- 4: " ERROR "
 TERMINATE(ATTRIBUTE);
- 6: " CALCULATION ::= EXPRESSION = "
 BEGIN
 OUTINTEGER(O\_STREAM, SEM\_STACK[STACKINDEX], O, CC);
 IF CC <> IO\_OK THEN TERMINATE(CC);
 FLUSH(O\_STREAM, CC);
 IF CC <> IO\_OK THEN TERMINATE(CC);
 END;
- 8: " EXPRESSION ::= EXPRESSION + TERM "
 SEM\_STACK[STACKINDEX]:= SEM\_STACK[STACKINDEX] + SEM\_STACK[STACKINDEX+2];
- 10: " TERM ::= TERM \* FACTOR "
 SEM\_STACK[STACKINDEX]:= SEM\_STACK[STACKINDEX] \* SEM\_STACK[STACKINDEX+2];
- 12: " FACTOR ::= ( EXPRESSION ) "
 SEM\_STACK[STACKINDEX]:= SEM\_STACK[STACKINDEX+1];
- 0, 1, 3, 5, 7, 9, 11: " NO ACTION "

```
END;
```

```
END;
```

```
$D**GENS.D*PARSE.D*PROCS.S
```

*on byte (O-stream, CH, CC)*  
*if CH = '\n' then*  
*line no := line no + 1*  
*write byte (O-stream, line no, # 0# 30, cc)*

BEGIN

```
" CONNECT CURRENT OUT FILE AND PARAMETER FILE "  
CONNECT(PARAM.OFILE, OUTPUT_MODE, O_STREAM, CC);  
IF CC <> IO_OK THEN TERMINATE(CC);  
CONNECT(PARAM.PFILE, INPUT_MODE, I_STREAM, CC);  
IF CC <> IO_OK THEN TERMINATE(CC);
```

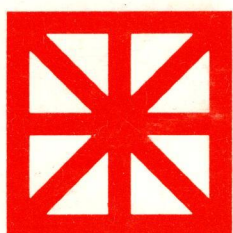
```
" SKIP FIRST LINE OF PARAMETER FILE AND READ TABLE FILE NAME "  
REPEAT IN_BYTE(CH) UNTIL CH = NL;  
FSN:= PARAM.FSN;  
VOL:= PARAM.VOL;
```

```
INFILEID(I_STREAM, FROM_ADAM, FSN, VOL, NAMELIST, NAMENO, CC);  
IF CC <> IO_OK THEN TERMINATE(CC);
```

```
" FIND TABLE FILE, CONNECT IT AND INITIALIZE PARSE TABLES "  
FIND_FILE(FROM_ADAM, FSN, VOL, NAMELIST, NAMENO, PARAM.OFILE, TABLEFILE, CC);  
IF CC <> IO_OK THEN TERMINATE(CC);  
DISCONNECT(I_STREAM, F, CC);  
IF CC <> IO_OK THEN TERMINATE(CC);  
CONNECT(TABLEFILE, INPUT_MODE, I_STREAM, CC);  
IF CC <> IO_OK THEN TERMINATE(CC);  
INIT_PARSE;  
DISCONNECT(I_STREAM, F, CC);  
IF CC <> IO_OK THEN TERMINATE(CC);
```

```
" CONNECT CURRENT INPUT FILE, PROMPT AND START PARSER "  
CONNECT(PARAM.IFILE, INPUT_MODE, I_STREAM, CC);  
IF CC <> IO_OK THEN TERMINATE(CC);  
OUTTEXT(O_STREAM, "READY(:10:)(:0:)", CC);  
FLUSH(O_STREAM, CC);  
IF CC <> IO_OK THEN TERMINATE(CC);  
PARSE;  
FLUSH(O_STREAM, CC);  
IF CC <> IO_OK THEN TERMINATE(CC);
```

END.



CHRISTIAN ROVSING A/S  
Copenhagen · Denmark