stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 100/78 NOVEMBER

R. GLANDORF, D. GRUNE & J. VERHAREN

A W-GRAMMAR OF ALEPH

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS(MOS) subject classification scheme (1970): 68A15

ACM-Computing Reviews-categories: 4.22, 4.12

A W-grammar of ALEPH

by

R. Glandorf, D. Grune & J. Verharen

ABSTRACT

A VW-grammar (2-level grammar) of the programming language ALEPH is presented. It comprises the context-free grammar as published before plus all static restrictions. Some comment is given on the problems encountered in writing and debugging a W-grammar. This paper should be read in parallel with the ALEPH Manual, since it by itself does not define or explain the language.

KEY WORDS & PHRASES: *ALEPH, W-grammar, grammar correctness, syntactic specification.*

1. About the grammar.

1.1. Introduction.

ALEPH (A Language Encouraging Program Hierarchy) is a programming language developed at the Mathematical Centre, Amsterdam and is based on the similarity between the grammatical description of a problem and the top-down program for solving it. It is used mainly for compiler writing.

ALEPH is described in the ALEPH Manual [1] (abbreviated AM) by means of a context-free grammar; restrictions and semantics are added in informal English. It is well known that when we switch from a context-free grammar to a context-sensitive grammar the restrictions, including even the identification process, can be incorporated in the grammar. The best form for such an effort is a 2-level grammar (also called van Wijngaarden grammar or VW-grammar), which is treated extensively by Cleaveland and Uzgalis [2]. The best known application of this technique to a large language can be found in the Revised Report on ALGOL68 [3].

The 2-level ALEPH grammar presented here comprises the old context-free grammar as given in the AM and all static restrictions described therein. It is recommended to read this paper side by side with the AM. Its structure corresponds generally to that of the AM. Discrepancies occur only where paragraphs in the AM have been combined (e.g., "3.3.2 actual affixes" of the AM is incorporated in "3.5. Affix forms" in this paper) or have been deleted, since they did not contain syntax in the AM.

Explanatory comment has been added where, in our opinion, the VW-grammar was hard to understand, or where a construction is used which differs considerably from that in the AM.

1.2. Correctness.

Some words must be said about debugging a VW-grammar. Writing a VW-grammar for a language is comparable to writing a checking parser for it in a language for which there is no compiler. So computer debugging fails and there are no standards for writing "structured" VW-grammars. In our style we followed the only example we had, the ALGOL 68 Revised Report. Here correctness was achieved by reading and rereading the grammar over a number of years.

We tried to use the same technique but soon found out that an 800 line VW-grammar exceeds our mental capacity and we were forced to look for some mechanical aids. Although many techniques and programs are available for analysing, cross-referencing and checking context-free-grammars, we know of no such technique or program for VW-grammars.

1.3. Debugging techniques.

We used the following techniques, aided by the text editing and manipulation features of the UNIX operating system [4].

- It was checked if the grammar conformed to the syntax of a VW-grammar, as specified in [2], page 47, and [1], 1.1.3.
- A cross-reference was made of all tags in the grammar. This brings together all NOTIONs that might be related, but it yielded such voluminous and repetitious output that it was of no use.
- The meta-grammar, which is context-free, was tested for consistency by the usual means. For each metanotion "declared" twice, it was checked that both declarations were identical.
- A hypernotion bears some resemblance to a procedure call with parameters, a hyperrule to a procedure declaration and a VW-grammar to a program. E.g., in 'NEST info sequence with NEST1' the procedure name is 'info sequence with' and NEST and NEST1 are the parameters.

  It is on this resemblance that ALEPH is based, and it is only natural to try to exploit this resemblance for checking purposes, i.e., to check consistency of parameter types, or metanotions. Such a test would be comparable to compile time testing in a typed language: it still does not guarantee that the program is correct, but it catches many errors. Unfortunately, in a VW-grammar, two parameters in a call can coalesce into one in the declaration (or vice versa), part of the procedure name can derive from a parameter, several different declarations may be present for the same name, etc.

We tried two checking heuristics. In both cases the hypergrammar was split into two lists of hypernotions, one containing all "declarations", i.e., hypernotions occurring in front of a colon, and one containing all "applications", i.e., hypernotions occurring after the colon. These lists were checked against each other and discrepancies reported. A first run yielded about 400 discrepancies.

In order to reduce this number we tried metanotion production, e.g., each entry containing "TYPE" was replaced by two entries, one with "fallible" and one with "infallible". This makes both lists much longer, but regardless of which metanotions we substituted, the number of discrepancies remained around 400.

We then tried metanotion reduction, e.g., each occurrence of the word "fallible" was replaced by "TYPE", since looking at the grammar had convinced us that "TYPE" was the only source of "fallible". We used only metanotions that produced a finite-choice language. The process reduced the number of discrepancies to 137, which were then checked by hand.

1.4. Sources of errors.

The most frequent error found was "slight textual mismatch between 'hypernotion declaration' and 'hypernotion application'", which is to be expected in an unchecked piece of text written by more than one person over a period of more than one year.

More important is the fact that the attractive features "NOTION option", "NOTION list", etc., as they appear in the ALGOL68 Report, were stubborn trouble makers, and should, in the jargon, be considered harmful, i.e., error prone. Their trouble is that they do not allow any kind of context condition. So, a "NOTION option" is really optional, regardless of anything, and such a situation seldom arises in a programming language. E.g., every ALEPH user knows that the 'selector' in front of an 'element' is optional, and the first line of '... element' in 3.5 read 'FIELDS selection option', until it dawned upon us that a missing selector is interpreted as being equal to the TAG of the 'table' or 'stack' concerned (AM 3.4.1). But this 'table' or 'stack' need not have a 'selector' TAG, and an incorrect program could be produced through this 'FIELDS selection option'!

In the end we only kept the "NOTION pack" and the "NOTION option".

1.5. Conclusion.

The conclusion seems to be that a VW-grammar is too general a tool to allow easy mechanical checking. Both the approach of restricting the VW-grammar and the approach of extensive mechanical checking look promising, but since this work is not on the checking of VW-grammars, these issues will not be pursued here any further.

1.6. References.

[1] Grune, D., R. Bosch and L.G.L.T. Meertens, ALEPH Manual, IW 17/75, Mathematical Centre, Amsterdam, 1975.

[2] Cleaveland, J.C., R.C. Uzgalis, Grammars for Programming Languages, Programming Language Series 4, Elsevier Scientific Publishing Company, Amsterdam, 1977.

[3] Wijngaarden, A. van, et al. (Eds), Revised Report on the Algorithmic Language ALGOL68, Acta Informatica, 5, 1-236, 1975.

[4] Ritchie, D.M., K. Thompson, The UNIX time-sharing system, Communications of the ACM, 17(1974)7, 365-375.

2. General rules.

NOTION:: ALPHA; NOTION ALPHA.
NOTETY:: NOTION; EMPTY.

```
THING:: NOTION; ( NOTETY1 ) NOTETY2; THING ( NOTETY1 ) NOTETY2.
WHETHER:: where; unless.
EMPTY:: .

where true:
    EMPTY.
unless false:
    EMPTY.
where THING1 and THING2:
    where THING1, where THING2.
where THING1 or THING2:
    where THING1; where THING2.
unless THING1 and THING2:
    unless THING1; unless THING2.
unless THING1 or THING2:
    unless THING1, unless THING2.

WHETHER ( NOTETY1 ) is ( NOTETY2 ):
    WHETHER ( NOTETY1 ) begins with ( NOTETY2 ) and
      ( NOTETY2 ) begins with ( NOTETY1 ).
WHETHER ( EMPTY ) begins with ( NOTION ):
    WHETHER false.
WHETHER ( NOTETY ) begins with ( EMPTY ):
    WHETHER true.
WHETHER ( ALPHA1 NOTETY1 ) begins with ( ALPHA2 NOTETY2 ):
    WHETHER ( ALPHA1 ) coincides with ( ALPHA2 ) in
      ( abcdefghijklmnopqrstuvwxyz )
      and ( NOTETY1 ) begins with ( NOTETY2 ).
where ( ALPHA ) coincides with ( ALPHA ) in ( NOTION ):
    where true.
unless ( ALPHA1 ) coincides with ( ALPHA2 ) in ( NOTION ):
    where ( NOTION ) contains ( ALPHA1 NOTETY ALPHA2 )
      or ( NOTION ) contains ( ALPHA2 NOTETY ALPHA1 ).
WHETHER ( ALPHA NOTETY ) contains ( NOTION ):
    WHETHER ( ALPHA NOTETY ) begins with ( NOTION ) or
      ( NOTETY ) contains ( NOTION ).
WHETHER ( EMPTY ) contains ( NOTION ):
    WHETHER false.

NOTION pack:
    open symbol,
    NOTION,
    close symbol.
NOTION option:
    EMPTY;
    NOTION.
```

These definitions have been copied from the ALGOL 68 Report [3]. Note that the 1's and 2's in the definition of THING have been retained although they are completely superfluous.

3. Program logic.

3.0. Nest.

```
NEST:: DECSETY.
DECSETY:: DECS; EMPTY.
DECS:: DEC; DECS DEC.
DEC:: DESCR TAG.
DESCR:: STACK; NONSTACK.
NONSTACK:: CONSTABLE; TABLE; FILE; RULE.
CONSTABLE:: CONSTANT; VARIABLE.
STABLE:: STACK; TABLE.
CONSTAGSETY:: CONSTAGS; EMPTY.
CONSTAGS:: CONSTAG; CONSTAGS CONSTAG.
CONSTAG:: CONSTANT TAG.
DEF:: TACK; constant; variable; rule; file; EXTERN.
TACK:: table; stack.
EXTERN:: external constant; external table;
    external TYPE EFFECT rule; standard.

TAG:: LETTER; TAG LETTER; TAG DIGIT.
LETTER:: letter ALPHA.
DIGIT:: digit NUMERIC.
ALPHA:: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q; r;
    s; t; u; v; w; x; y; z.
NUMERIC:: zero; one; two; three; four; five; six; seven;
    eight; nine.

LETTER: LETTER symbol.
TAG LETTER: TAG, LETTER symbol.
TAG DIGIT: TAG, DIGIT symbol.
```

3.1. Program.

```
program:
    where ( NEST ) is ( NEST1 NEST2 ),
    NEST info sequence with NEST1,
    NEST root,
    NEST info sequence with NEST2,
    end symbol.
NEST info sequence with EMPTY:
    EMPTY.
NEST info sequence with NEST1:
    NEST pragmat,
    NEST info sequence with NEST1.
NEST info sequence with NEST1 DECS:
    NEST DEF declaration of DECS,
    NEST info sequence with NEST1.

NEST root:
    root symbol,
    NEST new EMPTY TYPE EFFECT NOREQ affix form,
```

point symbol.

The definition of 'root' is in accordance with the AM, in which any 'TYPE EFFECT rule' is allowed in the 'root'. The present compiler requires the 'root' to contain an 'action' (for practical reasons).

3.2. Rules.

3.2.1. Rule declarations.

```
RULE:: TYPE EFFECT with PARAMSETY rule.
TYPE:: fallible; infallible.
EFFECT:: side; nonside.

NEST rule declaration of TYPE EFFECT with PARAMSETY rule TAG:
    TYPE EFFECT typer,
    NEST TYPE EFFECT with PARAMSETY rule
      TAG defining identifier,
    FORMALSETY req form FORMSETY PARAMSETY formals sequence,
    NEST new FORMALSETY inaccessible compound TYPE EFFECT
      label TAG LOCALSETY TYPE EFFECT
      req form FORMSETY req loc EMPTY actual rule,
    point symbol.

fallible nonside typer:
    question symbol.
fallible side typer:
    predicate symbol.
infallible nonside typer:
    function symbol.
infallible side typer:
    action symbol.
```

FORMSETY contains those formals that are used as output parameters; these must be initialized in the production of the 'rule'. For 'inaccessible compound' see 3.6.2.

3.2.2. Actual rules.

```
LF:: new FORMALSETY; LF BLOCK.
RULEBODY:: alternative series; classification.
MEMBIX:: member; affix.

NEST LF BLOCK TYPE EFFECT REQ actual rule:
    BLOCK REQ REQ1 local part,
    NEST LF BLOCK TYPE EFFECT REQ1 RULEBODY,
    where no doubles in LF BLOCK.
NEST LF TYPE EFFECT REQ alternative series:
    NEST LF TYPE1 TYPE2 EFFECT1 REQ1 alternative,
    NEST LF1 TYPE1 TYPE3 EFFECT2 REQ2 body tail,
    where EFFECT EFFECT1 EFFECT2 holds,
```

```
    where TYPE TYPE2 TYPE3 holds,
    where ( REQ1 ) ( LF1 REQ2 ) is alternative split of
      ( LF REQ ).
NEST LF TYPE TYPE nonside NOREQ body tail:
    EMPTY.
NEST LF fallible TYPE EFFECT REQ body tail:
    semicolon symbol,
    NEST LF TYPE EFFECT REQ alternative series.
NEST LF TYPE TYPE1 EFFECT REQ alternative:
    where ( TYPE1 ) is ( infallible ),
    NEST LF1 TYPE EFFECT REQ LASTMEMBER,
    where last block in LF is accessible in LF1;
    NEST LF TYPE EFFECT1 REQ1 MEMBER,
    NEST LF1 TYPE1 EFFECT2 REQ2 alternative tail,
    where EFFECT EFFECT1 EFFECT2 holds,
    where ( REQ1 ) ( LF1 REQ2 ) is member split of ( LF REQ ).
NEST LF TYPE EFFECT REQ alternative tail:
    comma symbol,
    NEST LF TYPE1 TYPE2 EFFECT REQ alternative,
    where TYPE TYPE1 TYPE2 holds.

where last block in LF inaccessible compound LABELETY LOCALSETY
  is accessible in LF accessible compound LABELETY LOCALSETY:
    where true.
where ( req form FORMSETY req loc LOCSETY1 )
  ( LF req form FORMSETY req loc LOCSETY2 )
  is alternative split of
  ( LF req form FORMSETY req loc LOCSETY ):
    where LOCALSETY is permutation of LOCALSETY1 LOCALSETY2.
where ( req form FORMSETY1 req loc LOCSETY1 )
  ( LF1 req form FORMSETY2 req loc LOCSETY2 )
  is MEMBIX split of
  ( LF req form FORMSETY req loc LOCSETY ):
    where FORMSETY is permutation of FORMSETY1 FORMSETY2,
    where LOCALSETY is permutation of LOCALSETY1 LOCALSETY2,
    where LF1 is MEMBIX equivalent with
      LF dot FORMSETY1 dot LOCALSETY1.
```

3.2.2.1. Alternative series.

The production of 'alternative series' starts by producing its first 'alternative'. The following items are recorded for this 'alternative': its type (in TYPE2); whether it has side effects (in EFFECT1); which locals must be initialized (in REQ1); and whether its first 'member' can fail (in TYPE1). Since the type of the first 'member' has its consequences for the 'body tail', TYPE1 is passed on to it. TYPE1 determines if the 'body tail' can be reached.

1. If TYPE1 is "infallible", the 'alternative' must be the last one. This is all right if TYPE3 is "infallible" too: the 'body tail' is then empty. If, however, TYPE3 is "fallible", then there is no production rule for 'body tail' and the requirements posed by 'where TYPE

TYPE2 TYPE3 holds' cannot be fulfilled.

2. If TYPE1 is "fallible" there are two cases to be distinguished. Case A: TYPE3 is "fallible"; the 'alternative' may or may not be the last and 'body tail' may be empty. Case B: TYPE3 is "infallible"; 'body tail' must be present.

3.2.2.2. Alternative.

Two types (TYPE and TYPE1) are recorded for an 'alternative'. TYPE is the type of the first 'member' and TYPE1 is the type of the rest of the 'alternative'.

3.2.2.3. Uniqueness of identifiers.

```
where no doubles in LF local INIT TAG:
    where TAG not in LF,
    where no doubles in LF.
where no doubles in LF COMPOUND TYPE EFFECT label TAG:
    where TAG not in LF.
where no doubles in LF COMPOUND:
    where TAG not in LF.
where TAG not in new EMPTY:
    where true.
where TAG not in LF local INIT TAG1:
    unless ( TAG ) is ( TAG1 ),
    where TAG not in LF.
where TAG not in LF COMPOUND TYPE EFFECT label TAG1:
    unless ( TAG ) is ( TAG1 ),
    where TAG not in LF.
where TAG not in LF COMPOUND:
    where TAG not in LF.
where TAG not in LF FDESCR TAG1:
    unless ( TAG ) is ( TAG1 ),
    where TAG not in LF.
```

This test ensures that the 'local TAG's differ, and moreover that each 'local TAG' differs from all other TAG's in LF.

3.2.2.4. Conservation of TYPE and EFFECT.

```
TEF:: TYPE; EFFECT.
TYPE:: fallible; infallible.
EFFECT:: side; nonside.

where TEF TEF1 TEF2 holds:
    where ( TEF ) is ( TEF1 ),
    where ( TEF ) is ( TEF2 );
    where ( TEF ) is ( fallible ) or ( TEF ) is ( side ),
    unless ( TEF1 ) is ( TEF2 ).
```

TYPE tells whether the NOTION can or cannot fail. EFFECT shows if it has side effects. TYPE is passed on as follows:

TYPE NOTION: TYPE1 NOTION1, TYPE2 NOTION2.

This yields the following possibilities:

TYPE	TYPE1	TYPE2	
fallible	fallible	infallible	(1)
f	i	f	(2)
f	f	f	(3)
f	i	i	(4)
i	f	f	(5)
i	f	i	(6)
i	i	f	(7)
i	i	i	(8)

If NOTION cannot fail (TYPE is "infallible"), NOTION1 and NOTION2 cannot fail either. Consequently (5), (6) and (7) are not allowed to occur. If NOTION can fail (TYPE is "fallible"), at least one of the other notions NOTION1 and NOTION2 must be able to fail. This eliminates (4).

The same scheme and the same exceptions apply to EFFECT. All this is checked by the rule 'where TEF holds'.

3.2.2.5. Requirements.

```
REQ:: req form FORMSETY req loc LOCSETY.
NOREQ:: req form EMPTY req loc EMPTY.
FORMSETY:: FORMS; EMPTY.
FORMS:: FORM; FORMS FORM.
FORM:: formal uninit TAG.
OBJECTSETY:: OBJECTS; EMPTY.
OBJECTS:: FIELDS; LOCS; FORMS.
OBJECT:: FIELD; LOC; FORM.

where OBJECTSETY1 OBJECT OBJECTSETY2 is permutation of
  OBJECTSETY3 OBJECT:
    where OBJECTSETY1 OBJECTSETY2 is permutation of
      OBJECTSETY3.
where EMPTY is permutation of EMPTY:
    where true.

where NOTION1 local INIT TAG NOTION2 is MEMBIX equivalent with
  NOTION3 local uninit TAG NOTION4
  dot FORMSETY dot LOCALSETY local uninit TAG:
    where MEMBIX split sets to INIT,
    where NOTION1 NOTION2 is MEMBIX equivalent with
      NOTION3 NOTION4 dot FORMSETY dot LOCALSETY.
where NOTION1 formal INIT TAG NOTION2 is MEMBIX equivalent with
  NOTION3 formal uninit TAG NOTION4
```

```
  dot FORMSETY formal uninit TAG dot EMPTY:
    where MEMBIX split sets to INIT,
    where NOTION1 NOTION2 is MEMBIX equivalent with
      NOTION3 NOTION4 dot FORMSETY dot EMPTY.
where NOTION is MEMBIX equivalent with
  NOTION dot EMPTY dot EMPTY:
    where true.
where member split sets to init:
    where true.
where affix split sets to halfinit:
    where true.
```

REQ contains a list of formals and locals to be initialized. The following restrictions apply:
- all formals occurring in REQ must be initialized by the end of each 'alternative',
- all locals occurring in REQ must be used (i.e. initialized) at least once.

If the production of an 'alternative series', a 'classification' or an 'affix sequence' fulfils such an initialization requirement this must be recorded in LF. This LF can be considered as a "run-time stack" in which the initialization states of the formals and locals are kept rather than their actual values. The initial state for all formals and locals occurring in REQ is "uninit". When a requirement from REQ is fulfilled, the state of the formal or local must be changed to "halfinit" or "init". This change is effected by the rules 'is member equivalent with' and 'is affix equivalent with' which set to "init" and "halfinit" respectively.

In AM 3.3.3 it is required that each local must be used in at least one 'alternative', and it then requires that this 'alternative' does not end in a 'jump', 'exit' or 'failure symbol'. This second restriction is considered unjustified and is not implemented in the present grammar.

3.2.3. Members.

```
MEMBER:: affix form; compound member; OPERATION.
LASTMEMBER:: MEMBER; TERMINATOR.
```

The metanotion LASTMEMBER has been introduced to restrict a TERMINATOR to the last 'member' of an 'alternative'.

3.3. Affixes.

3.3.1. Formal affixes.

```
LF:: new FORMALSETY; LF BLOCK.
FORMALSETY:: FORMALS; EMPTY.
FORMALS:: FORMAL; FORMALS FORMAL.
```

```
FORMAL:: FDESCR TAG.
FDESCR:: formal FITACK; formal INIT.
FITACK:: FITAB; FIELDS stack.
FITAB:: file; FIELDS table.

FORMAL FORMALSETY req form FORMSETY FORMSETY1
  PARAM PARAMSETY formals sequence:
    formal affix symbol,
    FORMAL req form FORMSETY PARAM formal,
    FORMALSETY req form FORMSETY1 PARAMSETY formals sequence.
EMPTY req form EMPTY EMPTY formals sequence:
    EMPTY.
formal file TAG req form EMPTY file parameter formal:
    quote image,
    formal file TAG defining identifier.
formal init TAG req form EMPTY in parameter formal:
    right symbol,
    formal init TAG defining identifier.
formal uninit TAG req form formal uninit TAG
  out parameter formal:
    formal uninit TAG defining identifier,
    right symbol.
formal init TAG req form EMPTY inout parameter formal:
    right symbol,
    formal init TAG defining identifier,
    right symbol.
formal FIELDS TACK TAG req form EMPTY
  TACK CAL parameter formal:
    TACK sub bus,
    FIELDS list of TACK TAG,
    formal FIELDS TACK TAG defining identifier, sub bus,
    where FIELDS has calibre CAL.

table sub bus:
    EMPTY.
stack sub bus:
    sub bus.
sub bus:
    sub symbol,
    bus symbol.

where FIELDSETY FIELD has calibre CALETY i:
    where FIELDSETY has calibre CALETY.
where EMPTY has calibre EMPTY:
    where true.
```

This test ensures that the number of selectors specified (CAL) correspond to the number of selectors in the declaration (FIELDS).

3.3.3. Local affixes.

```
BLOCKSETY:: BLOCKS; EMPTY.
BLOCKS:: BLOCK; BLOCKS BLOCK.
BLOCK:: COMPOUND LABELETY LOCALSETY.
COMPOUND:: ACCESS compound.
ACCESS:: accessible; inaccessible.
LABELETY:: LABEL; EMPTY.
LABEL:: TYPE EFFECT label TAG.

LOCALSETY:: LOCALS; EMPTY.
LOCALS:: LOCAL; LOCALS LOCAL.
LOCAL:: local INIT TAG.
INIT:: init; halfinit; uninit.
LOCSETY:: LOCS; EMPTY.
LOCS:: LOC; LOCS LOC.
LOC:: local uninit TAG.

inaccessible compound EMPTY EMPTY REQ REQ local part:
    EMPTY.
inaccessible compound LABEL LOCALSETY REQ REQ1 local part:
    LOCALSETY REQ REQ1 local part tail.
inaccessible compound EMPTY LOCALS REQ REQ1 local part:
    LOCALS REQ REQ1 local part tail.
LOCAL LOCALSETY REQ REQ1 LOC local part tail:
    local affix symbol,
    LOCAL req loc LOC local,
    LOCALSETY REQ REQ1 local part tail.
local uninit TAG req loc local uninit TAG local:
    local TAG defining identifier.
EMPTY REQ REQ local part tail:
    colon symbol.
```

The 'label TAG', if present, is produced upon the creation of the new BLOCK, i.e., through 'compound member' (see 3.7). The rule '... TAG local' ensures that all locals will be introduced in LF as "uninit" and that a copy will be entered into REQ.

3.4. Operations.

```
OPERATION:: transport; identity; extension.

NEST LF infallible EFFECT REQ transport:
    NEST LF source,
    NEST LF EFFECT REQ destination sequence with PARAMS,
    where PARAMS is correct for destinations.
NEST LF source:
    NEST LF nonside NOREQ form with in parameter.

NEST LF fallible nonside NOREQ identity:
    NEST LF source,
```

```
    equals symbol,
    NEST LF source.

NEST LF infallible side NOREQ extension:
    of symbol,
    NEST LF FIELDS field transport list,
    of symbol,
    NEST LF FIELDS stack TAG applied identifier.
NEST LF FIELDS field transport list:
    NEST LF FIELDS1 field transport,
    NEST LF FIELDSETY2 field transport list tail,
    where FIELDS is permutation of FIELDS1 FIELDSETY2.
NEST LF EMPTY field transport list tail:
    EMPTY.
NEST LF FIELDS field transport list tail:
    comma symbol,
    NEST LF FIELDS field transport list.
NEST LF FIELDS field transport:
    NEST LF source,
    FIELDS field destination sequence.
FIELD FIELDSETY field destination sequence:
    to token, FIELD tag,
    FIELDSETY field destination sequence.
EMPTY field destination sequence:
    EMPTY.
SELECTORSETY1 selector TAG SELECTORSETY2 field tag:
    TAG.
to token:
    minus symbol,
    right symbol.

where PARAMSETY out parameter is correct for destinations:
    where PARAMSETY is correct for destinations.
where EMPTY is correct for destinations:
    where true.
```

The above ensures the PARAMSETY in a 'destination sequence' to contain out-parameters only.

For 'where ... is permutation of ...' see 3.2.2.5.

An 'extension' is used to extend a 'stack'. The values to be put on the stack must be specified right away, but not necessarily in the right order. The following requirements hold:

- all fields must be filled,
- no field may be filled more than once.

3.5. Affix forms.

```
AFFIX:: actual affix; destination.
PARAMSETY:: PARAMS; EMPTY.
PARAMS:: PARAM; PARAMS PARAM.
PARAM:: INOUT parameter; file parameter; TACK CAL parameter.
```

```
TACK:: table; stack.
FORLOC:: formal; local.
INDIR:: index; direct.
INOUT:: in; inout; out.
INIT:: init; halfinit; uninit.

NEST LF TYPE EFFECT REQ affix form:
    NEST global TYPE EFFECT1 with PARAMSETY rule
      TAG applied identifier,
    NEST LF EFFECT2 REQ actual affix sequence with PARAMSETY,
    where EFFECT EFFECT1 EFFECT2 holds.
NEST LF EFFECT REQ AFFIX sequence with PARAM PARAMSETY:
    AFFIX token,
    NEST LF EFFECT1 REQ1 form with PARAM,
    NEST LF1 EFFECT2 REQ2 AFFIX sequence with PARAMSETY,
    where EFFECT EFFECT1 EFFECT2 holds,
    where ( REQ1 ) ( LF1 REQ2 ) is affix split of ( LF REQ ).
actual affix token:
    actual affix symbol.
destination token:
    to token.
NEST LF nonside NOREQ AFFIX sequence with EMPTY:
    EMPTY.

NEST LF nonside NOREQ form with file parameter:
    NEST LF file TAG applied identifier.
NEST LF nonside NOREQ form with TACK CAL parameter:
    NEST LF FIELDS TACK1 TAG applied identifier,
    where ( TACK ) is ( TACK1 ) or ( TACK1 ) is ( stack ),
    where FIELDS has calibre CAL.
NEST LF nonside NOREQ form with in parameter:
    NEST global CONSTABLE TAG applied identifier;
    PLAIN denotation;
    NEST LF TACK LIMIT;
    NEST LF in direct TACK element;
    LF FORLOC TAG in direct identifier.
NEST LF side NOREQ form with inout parameter:
    NEST global variable TAG applied identifier;
    NEST LF inout direct stack element.
NEST LF nonside NOREQ form with inout parameter:
    LF FORLOC TAG inout direct identifier.
NEST LF side NOREQ form with out parameter:
    NEST global variable TAG applied identifier;
    NEST LF out direct stack element.
NEST LF nonside NOREQ form with out parameter:
    LF FORLOC TAG out direct identifier;
    dummy symbol.
NEST LF nonside req form formal uninit TAG req loc EMPTY
  form with out parameter:
    LF formal TAG out direct identifier.
NEST LF nonside req form EMPTY req loc local uninit TAG
  form with out parameter:
```

```
    LF local TAG out direct identifier.

NEST LF INOUT INDIR TACK element:
    FIELDS selection from TAG,
    NEST LF FIELDS TACK TAG applied identifier,
    sub symbol,
    NEST LF INOUT index,
    bus symbol,
    where ( INDIR ) is ( index ) or ( TACK ) is ( stack )
      or ( INOUT ) is ( in ).
FIELDSETY1 SELECTORSETY1 selector TAG SELECTORSETY2 FIELDSETY2
  selection from TAG1:
    where ( TAG ) is ( TAG1 ), EMPTY;
    TAG, of symbol.
NEST LF INOUT index:
    NEST global CONSTABLE TAG applied identifier;
    PLAIN denotation;
    NEST LF TACK LIMIT;
    NEST LF INOUT index TACK element;
    LF FORLOC TAG INOUT index identifier.
LF FORLOC TAG INOUT INDIR identifier:
    LF FORLOC init TAG applied identifier;
    where ( INOUT ) is ( out ),
    LF FORLOC halfinit TAG applied identifier;
    where ( INOUT ) is ( out ) and ( INDIR ) is ( direct ),
    LF FORLOC uninit TAG applied identifier.
```

3.5.1. Parameter checking.

There are 5 kinds of parameters: file-, TACK-CAL-, in-, inout- and out-parameters. These 5 kinds of parameters fall in two categories:

- "Static" parameters,
- "Transport" parameters.

"Static" parameters are the file- and TACK-CAL-parameters, for which it is required that the name (TAG) be present as global or formal. For the TACK-CAL-parameter it is also required that the number of selectors specified (CAL) correspond to the number of selectors (FIELDS) in the global or formal declaration.

"Transport" parameters are the in-, inout- and out-parameters.
An in-parameter can be a constant, a variable, a denotation, a limit, an element, a formal or a local.
An inout-parameter can be a variable, an element, a formal or a local.
An out-parameter can be a variable, an element, a formal, a local or dummy.

Values may be copied from and/or to these parameters according to rules laid down in the AM 3.4.1. This Manual specifies parameter transfer to take place in 2 phases:
- in- and inout-parameters are copied before the actual rule call,
- inout- and out-parameters are copied back after the actual rule call.

It is required that the source be "reachable" and "initialized", and that the destination be "reachable". Normally a source or a destination is "reachable", except when it is an element the index of which is an uninitialized formal or local. Variables, constants, destinations and limits are always "reachable" and "initialized"; elements, if "reachable", are always "initialized"; formals and locals (forlocs for short) are always "reachable".

3.5.2. Initialization test.

We shall first consider the "initialized"-ness of forlocs, since the reachability of elements depends on it. The test for forlocs in phase 1 follows directly from the AM and is summarized in Table 1. In phase 2 there are no requirements for the forlocs.

Table 1.

		INIT	
		uninit	init
INOUT	in	-	+
	out	+	+
	inout	-	+

Here INOUT is the type of the formal parameter and INIT is the state of the forloc which is the actual parameter.

3.5.3. Reachability of elements.

Reachability is required in phase 1 for elements used as in- or inout-parameters, and in phase 2 for elements used as inout- and out-parameters. In both cases the requirement implies that a forloc in the index be initialized. In phase 1 this is only possible if the forloc was already initialized; in phase 2 initialization may have been done by a preceding out-parameter.

Example:
If "a" is declared as:

```
'action' a + b> + c>: ....
```

then

```
(-p: a + p + b[p], ...)
```

is correct:

	p	b[p]
phase 1:	uninit	unreach.
phase 2:	init	reachable

However,

(-p: a + b[p] + p, ...)
is incorrect:

	b[p]	p
phase 1:	unreach.	uninit
phase 2:	unreach.!	init

Now a two-phase test would normally result in two checking rules, but in this grammar we want to test while producing the text. So we have to devise a one-phase test.

In the two-phase approach each forloc has two possible states. In both phases the state can be "init" or "uninit". This yields 4 combinations:

	(1)	(2)	(3)	(4)
phase 1:	init	uninit	uninit	init
phase 2:	init	init	uninit	uninit

Now an initialized forloc can never decay into an uninitialized one in ALEPH, so combination (4) is rejected. This leaves us three combinations, which are our new states. We shall call combination (2) "halfinit".

Table 1 now extends to:

Table 2

		INIT		
		uninit	halfinit	init
INOUT	in	-	-	+
	out	+	+	+
	inout	-	-	+

Here again INOUT is is the type of the formal parameter and INIT is the state of the forloc which is the actual parameter.

The test for the reachability of an element is then summarized in the following table:

Table 3

		INIT		
		uninit	halfinit	init
INOUT	in	-	-	+
	out	-	+	+
	inout	-	-	+

Here INOUT is the way the element is used and INIT is the state of the forloc in the index.

Since, by pure coincidence, Tables 2 and 3 are so similar, we have combined them into one rule. The metanotion INDIR decides which table to use.

3.6. Terminators.

```
TERMINATOR:: failure; success; exit; jump.
FORMREQ:: req form FORMSETY req loc EMPTY.
```

```
NEST LF fallible nonside FORMREQ failure:
    failure symbol.
NEST LF infallible nonside NOREQ success:
    success symbol.
NEST LF infallible side FORMREQ exit:
    exit symbol,
    NEST free expression.
NEST LF TYPE EFFECT FORMREQ jump:
    repeat symbol,
    LF label TYPE EFFECT label TAG applied identifier,
    where TAG is permitted in LF.

where TAG is permitted in LF LOCALSETY:
    where TAG is permitted in LF.
where TAG is permitted in
  LF accessible compound TYPE EFFECT label TAG1:
    where ( TAG ) is ( TAG1 );
    where TAG is permitted in LF.
where TAG is permitted in LF accessible compound EMPTY:
    where TAG is permitted in LF.
```

3.6.1. Requirements on formals.

In AM 3.3.1 it is required that output formals must be "initialized" at the end of each 'alternative' except when the 'alternative' ends in a 'jump', 'exit' or 'failure symbol'. This exception is implemented by FORMREQ.

3.6.2. Accessibility.

A 'terminator' is only permitted in a LASTMEMBER. For a 'jump' to be permitted as a 'terminator' two requirements must be fulfilled:

The TAG following the 'repeat symbol' must be the TAG of the 'actual rule' or of a 'compound member' and must occur as such in LF, with the correct TYPE and EFFECT. This is checked by 'applied identifier'.

If 'applied identifier' indeed finds the label TAG, it must still be checked if the label TAG is "reachable".

For the 'actual rule' and for each 'compound member' a new BLOCK is added to LF, consisting of 'inaccessible compound LABELETY LOCALSETY'. Here ACCESS (:: inaccessible; accessible) is used as a toggle which indicates whether the label TAG is accessible for a 'jump'. This toggle is switched to "accessible" at the beginning of a LASTMEMBER.

The predicate 'where TAG is permitted in' checks if there is an 'inaccessible compound' on the way to the given label TAG. If such is the case, it implies that the 'jump' occurs in a 'member' which is not a LASTMEMBER in some intermediate 'compound member'; and this disallows the 'jump'.

```
    Example (compound member):
        (label0: ..., ..., #A(11: ..., #B ...;
                                  ..., #C ...;
                                  ...,   ..., (12: ..., #D ...;
                                                   ..., #E ...;
                                                   ..., #F ...
                                               ), #G ...
                              )
        );
```

For the position #A through #G accessibility and reachability are as follows:

Position	accessible	reachable for jump
#A	label0	label0
#B	label0, l1	label0, l1
#C	label0, l1	label0, l1
#D	label0, 12	12 (l1 being inaccessible)
#E	label0, 12	12 (" " ")
#F	label0, 12	12 (" " ")
#G	label0, l1	label0, l1.

3.7. Compound members.

```
NEST LF TYPE EFFECT REQ compound member:
    open symbol,
    BLOCK TYPE EFFECT label addition,
    NEST LF BLOCK TYPE EFFECT REQ actual rule,
    close symbol.
inaccessible compound EMPTY LOCALSETY
  TYPE EFFECT label addition:
    EMPTY.
inaccessible compound TYPE EFFECT label TAG LOCALSETY
  TYPE EFFECT label addition:
    TAG.
```

If the label TAG is present, it differs from all locals in the newly formed BLOCK and from all TAG's in LF (3.2.2.3).

3.8. Classifications.

```
NEST LF TYPE EFFECT REQ classification:
    NEST LF class box,
    NEST LF TYPE EFFECT REQ class series.
NEST LF class box:
    box symbol,
    NEST LF source,
    box symbol.
NEST LF TYPE EFFECT REQ class series:
    NEST LF TYPE1 TYPE2 EFFECT1 REQ1 class,
    NEST LF1 TYPE1 TYPE3 EFFECT2 REQ2 class series tail,
    where TYPE TYPE2 TYPE3 holds,
```

```
    where EFFECT EFFECT1 EFFECT2 holds,
    where ( REQ1 ) ( LF REQ2 ) is alternative split of
      ( LF REQ ).
NEST LF TYPE infallible nonside NOREQ class series tail:
    EMPTY.
NEST LF fallible TYPE EFFECT REQ class series tail:
    semicolon symbol,
    NEST LF TYPE EFFECT REQ class series.
NEST LF TYPE TYPE1 EFFECT REQ class:
    NEST TYPE area addition,
    NEST LF TYPE2 TYPE3 EFFECT REQ alternative,
    where TYPE1 TYPE2 TYPE3 holds.

NEST infallible area addition:
    EMPTY.
NEST fallible area addition:
    NEST area,
    comma symbol.
NEST area:
    sub symbol,
    NEST zone series,
    bus symbol.
NEST zone series:
    NEST zone,
    NEST zone series tail option.
NEST zone series tail:
    semicolon symbol,
    NEST zone series.
NEST zone:
    NEST free expression option,
    up to symbol,
    NEST free expression option;
    NEST free expression;
    NEST global STABLE TAG applied identifier.
```

A 'classification' is like an 'alternative series' in that both specify a row of 'alternative's. The difference is, that in a 'classification' there is always one 'alternative' ('class') which is chosen for execution, whereas in an 'alternative series' all 'alternative's may fail. The explanation given for 'alternative series' in 3.2.2.1 is also valid for a 'classification' (replace 'body tail' by 'class series tail'), except when TYPE1 is "fallible".

If TYPE1 and TYPE3 are both "fallible" the 'class series tail' cannot be empty, as a consequence of the fact that one 'class' must always be chosen.

If TYPE1 is "fallible" but TYPE3 is not, the 'class' may or may not be the last and 'class series tail' may be empty.

4. Data.

4.1. Integer based data.

4.1.1. Expressions.

```
RESTRICTED:: free; DEPENDSETY.
PLAIN:: integral; character.

NEST RESTRICTED expression:
    plus minus option,
    NEST RESTRICTED term;
    NEST RESTRICTED expression,
    plus minus,
    NEST RESTRICTED term.
NEST RESTRICTED term:
    NEST RESTRICTED term head option,
    NEST RESTRICTED base.
NEST RESTRICTED term head:
    NEST RESTRICTED term,
    times by.
NEST RESTRICTED base:
    NEST RESTRICTED plain value,
    NEST RESTRICTED expression pack.
NEST RESTRICTED plain value:
    PLAIN denotation;
    NEST global DEPENDSETY constant TAG applied identifier,
    where TAG included in RESTRICTED
      or ( RESTRICTED ) is ( free );
    NEST new EMPTY table LIMIT.

integral denotation:
    integral denotation option, DIGIT symbol.
character denotation:
    absolute symbol,
    NOTION symbol,
    absolute symbol.
plus minus:
    plus symbol;
    minus symbol.
times by:
    times symbol;
    by symbol.

where TAG included in DEPENDSETY1 dependent on TAG DEPENDSETY2:
    where true.
```

Two kinds of 'expression's must be distinguished.
First: 'expression's whose 'constant TAG's are restricted to those in DEPENDSETY. The predicate 'where TAG included in DEPENDSETY' takes care of this.

Second: 'expression's in which 'constant TAG's may occur from the entire NEST.

4.1.2. Constants.

```
CONSTANT:: DEPENDSETY constant.
DEPENDSETY:: DEPENDS; EMPTY.
DEPENDS:: DEPEND; DEPENDS DEPEND.
DEPEND:: dependent on TAG.

NEST constant declaration of DECS:
    constant symbol,
    NEST constant description list of DECS,
    point symbol.
NEST constant description list of
  DECSETY DEPENDSETY constant TAG:
    where DEPENDSETY is finite in NEST,
    NEST DEPENDSETY constant TAG defining identifier,
    equals symbol,
    NEST DEPENDSETY expression,
    NEST constant description list tail of DECSETY.
NEST DEF description list tail of EMPTY:
    EMPTY.
NEST DEF description list tail of DECS:
    comma symbol,
    NEST DEF description list of DECS.

where DEPENDSETY dependent on TAG is finite in NEST:
    where ( NEST ) is ( NEST1 DEPENDSETY1 constant TAG NEST2 ),
    where DEPENDSETY1 is finite in NEST,
    where DEPENDSETY is finite in NEST.
where EMPTY is finite in NEST:
    where true.
```

DEPENDSETY contains a list of 'constant TAG's on which the 'constant TAG' is directly dependent. The predicate 'where DEPENDSETY is finite in NEST' ensures that no circular 'constant declaration's can occur.

This check uses the following technique. Consider the declaration of a 'constant TAG'. This 'constant TAG' may be directly dependent of a number of other 'constant TAG's which form its DEPENDSETY. We now consider the rightmost 'constant TAG1' in this DEPENDSETY and find that it has its own DEPENDSETY1 in the NEST. By the same token we consider the rightmost 'constant TAG2' in DEPENDSETY1, which has its own DEPENDSETY2 in the NEST, etc, etc. This process terminates if and only if each TAG in DEPENDSETY can obtain a value in a finite number of steps. Circular 'constant declaration's are effectively prevented, since the production tree needed to produce them would be infinitely large.

4.1.3. Variables.

```
VARIABLE:: variable.

NEST variable declaration of DECS:
    variable symbol,
    NEST variable description list of DECS,
    point symbol.
NEST variable description list of DECSETY variable TAG:
    NEST variable TAG defining identifier,
    equals symbol,
    NEST free expression,
    NEST variable description list tail of DECSETY.
```

4.1.5. Tables.

```
TABLE:: FIELDS table.
FIELDS:: FIELD; FIELDS FIELD.
FIELDSETY:: FIELDS; EMPTY.
FIELD:: SELECTORS field.
SELECTORS:: SELECTOR; SELECTORS SELECTOR.
SELECTORSETY:: SELECTORS; EMPTY.
SELECTOR:: selector TAG.
TACK:: table; stack.

NEST TACK declaration of DECS:
    TACK symbol,
    NEST TACK description list of DECS,
    point symbol.
NEST table description list of
  DECSETY CONSTAGSETY FIELDS table TAG:
    FIELDS list of table TAG,
    NEST FIELDS table TAG defining identifier,
    where FIELDS has calibre CAL,
    NEST CONSTAGSETY EMPTY filling head CAL,
    NEST table description list tail of DECSETY.

FIELDS list of TACK TAG:
    where ( FIELDS ) is ( selector TAG field ), EMPTY;
    FIELDS definition list pack.
FIELD FIELDSETY definition list:
    where FIELD not the same tags as FIELDSETY,
    FIELD definition,
    FIELDSETY definition list tail.
EMPTY definition list tail:
    EMPTY.
FIELDS definition list tail:
    comma symbol,
    FIELDS definition list.
SELECTOR SELECTORSETY field definition:
    where SELECTOR no part of SELECTORSETY field,
    SELECTOR definition,
```

```
    SELECTORSETY field definition tail.
EMPTY field definition tail:
    EMPTY.
SELECTORS field definition tail:
    equals symbol,
    SELECTORS field definition.
selector TAG definition:
    TAG.

where FIELD not the same tags as EMPTY:
    where true.
where selector TAG SELECTORSETY field
  not the same tags as FIELDS:
    where selector TAG no part of FIELDS,
    where SELECTORSETY field not the same tags as FIELDS.
where EMPTY field not the same tags as FIELDS:
    where true.
where selector TAG no part of FIELDS FIELD:
    where selector TAG no part of FIELD,
    where selector TAG no part of FIELDS.
where selector TAG no part of selector TAG1 SELECTORSETY field:
    unless ( TAG ) is ( TAG1 ),
    where selector TAG no part of SELECTORSETY field.
where selector TAG no part of EMPTY field:
    where true.
```

This test ensures that all TAG's in a 'FIELD definition list' are different.

4.1.6. Stacks.

```
STACK:: DEPENDSETY FIELDS stack.
CALETY:: CAL; EMPTY.
CAL:: i; CAL i.
SIZE:: relative; absolute; absent.

NEST stack description list of
  DECSETY CONSTAGSETY DEPENDSETY FIELDS stack TAG:
    NEST DEPENDSETY SIZE size estimate,
    where all estimates in NEST have DEPENDSETY,
    FIELDS list of stack TAG,
    NEST DEPENDSETY FIELDS stack TAG defining identifier,
    where FIELDS has calibre CAL,
    NEST CONSTAGSETY DEPENDSETY filling head CAL with SIZE,
    NEST stack description list tail of DECSETY.
NEST CONSTAGSETY DEPENDSETY filling head CAL with SIZE:
    where ( CONSTAGSETY ) is ( EMPTY ),
    unless ( SIZE ) is ( absent ),
    EMPTY;
    NEST CONSTAGSETY DEPENDSETY filling head CAL.
```

```
NEST CONSTAGSETY DEPENDSETY filling head CAL:
    equals symbol,
    NEST DEPENDSETY filling CAL list CONSTAGSETY pack.
NEST DEPENDSETY filling CAL list CONSTAGSETY:
    NEST filling CAL,
    NEST DEPENDSETY filling CAL list CONSTAGSETY tail.
NEST DEPENDSETY filling CAL list
  CONSTAGSETY DEPENDSETY constant TAG:
    NEST filling CAL,
    colon symbol,
    where DEPENDSETY is finite in NEST,
    NEST DEPENDSETY constant TAG defining identifier,
    NEST DEPENDSETY filling CAL list CONSTAGSETY tail.
NEST DEPENDSETY filling CAL list CONSTAGSETY tail:
    comma symbol,
    NEST DEPENDSETY filling CAL list CONSTAGSETY.
NEST DEPENDSETY filling CAL list EMPTY tail:
    EMPTY.

NEST filling i:
    string denotation;
    NEST free expression.
NEST filling CAL i:
    NEST free expression list CAL i pack.
NEST free expression list CALETY i:
    NEST free expression,
    NEST free expression list tail CALETY.
NEST free expression list tail EMPTY:
    EMPTY.
NEST free expression list tail CAL:
    comma symbol,
    NEST free expression list CAL.
string denotation:
    quote symbol,
    string item sequence option,
    quote symbol.
string item sequence:
    string item, string item sequence option.
string item:
    unless ( NOTION ) is ( quote ), NOTION symbol;
    quote image.
quote image:
    quote symbol, quote symbol.

NEST DEPENDSETY relative size estimate:
    sub symbol,
    NEST DEPENDSETY expression,
    bus symbol.
NEST DEPENDSETY absolute size estimate:
    sub symbol,
    box symbol,
    NEST DEPENDSETY expression,
```

```
    box symbol,
    bus symbol.
NEST DEPENDSETY absent size estimate:
    EMPTY.
```

Here 'size estimate' passes its DEPENDSETY on to 'NEST DEPENDSETY expression' which will restrict the 'constant TAG's occurring in the 'expression' to these in DEPENDSETY.

4.1.6.1. Circularity through 'stack declaration's.

```
where all estimates in DECSETY NONSTACK TAG have DEPENDSETY:
    where all estimates in DECSETY have DEPENDSETY.
where all estimates in DECSETY DEPENDSETY FIELDS stack TAG
  have DEPENDSETY:
    where all estimates in DECSETY have DEPENDSETY.
where all estimates in EMPTY have DEPENDSETY:
    where true.
```

The location of a stack in the virtual address space (see AM 4.1.1) is determined by the lengths of the tables and stacks without 'size estimate' on one hand, and by the 'size estimate's of those stacks with 'size estimate's on the other hand. Since in the first case there are no 'size estimate's, there are no 'constant TAG's and there cannot be circularities. In the second case, however, there are 'size estimate's which contain 'expression's which may contain 'constant TAG's. Now 'pointer initialization's declare 'constant TAG's, which may be the cause of circularities, either directly or through the 'size estimate's of other stacks.

To prevent this we construct a DEPENDSETY which contains all 'constant TAG's in the 'size estimate's of all stacks, and pass this DEPENDSETY to each stack. This is implemented by checking in each 'stack declaration' that its DEPENDSETY is equal to the DEPENDSETY's of all stacks in the NEST. Again it is required that each TAG in DEPENDSETY can get a value in a finite number of steps (see 4.1.2).

This reveals a minor trouble spot in the AM. It is specified (in AM 4.1.4.d) that the "remainder of the virtual address space is distributed over the rest of the stacks, proportionally to their 'relative size's", but it is not specified in what order. Since the value of a 'constant TAG' in a 'pointer initialization' will depend only on the 'relative size's of the stacks treated earlier, ALEPH programs can be constructed in which changing the order of declarations might affect its correctness. The grammar given here takes the safe view that all 'constant TAG's in 'pointer initialization's are dependent on all 'relative size's.

4.1.7. Limits.

```
MICA:: max; min; calibre.
LIMIT:: MICA limit.

min token:
    left symbol, left symbol.
max token:
    right symbol, right symbol.
calibre token:
    left symbol, right symbol.
NEST LF TACK MICA limit:
    MICA token,
    NEST LF FIELDS TACK TAG applied identifier.
```

4.2. File declarations.

```
FILE:: file.

NEST file declaration of DECS:
    file typer,
    NEST file description list of DECS,
    point symbol.
NEST file description list of DECSETY file TAG:
    NEST file TAG defining identifier,
    NEST area option,
    equals symbol,
    right symbol option,
    string denotation,
    right symbol option,
    NEST file description list tail of DECSETY.
file typer:
    charfile symbol;
    datafile symbol.
```

5. Externals.

5.1. External declarations.

```
EXTERN:: external constant; external table;
    external TYPE EFFECT rule; standard.
```

5.2.1. External constants.

```
NEST external constant declaration of DECS:
    external symbol,
    constant symbol,
    NEST external constant description list of DECS,
    point symbol.
```

```
NEST external constant description list of
  DECSETY EMPTY constant TAG:
    NEST EMPTY constant TAG defining identifier,
    equals symbol,
    string denotation,
    NEST external constant description list tail of DECSETY.
```

5.2.2. External tables.

```
NEST external table declaration of DECS:
    external symbol,
    table symbol,
    NEST external table description list of DECS,
    point symbol.
NEST external table description list of
  DECSETY FIELDS table TAG:
    FIELDS list of table TAG,
    NEST FIELDS table TAG defining identifier,
    equals symbol,
    string denotation,
    NEST external table description list tail of DECSETY.
```

5.2.3. External rules.

```
NEST external TYPE EFFECT rule declaration of DECS:
    external symbol,
    TYPE EFFECT typer,
    NEST external TYPE EFFECT rule description list of DECS,
    point symbol.
NEST external TYPE EFFECT rule description list of
  DECSETY TYPE EFFECT with PARAMSETY rule TAG:
    NEST TYPE EFFECT with PARAMSETY rule TAG
      defining identifier,
    FORMALSETY req form FORMSETY PARAMSETY formals sequence,
    equals symbol,
    string denotation,
    NEST external TYPE EFFECT rule description list tail
      of DECSETY.
```

5.3. Standard externals.

```
STANDARD:: infallible nonside with in parameter in parameter
      out parameter out parameter rule
      letter a letter d letter d;
    fallible nonside with in parameter in parameter rule
      letter l letter e letter s letter s;
    infallible side with
      in parameter in parameter out parameter rule
      letter r letter a letter n letter d letter o letter m;
    fallible side with file parameter stack i parameter
      out parameter rule letter g letter e letter t letter l
```

```
      letter i letter n letter e;
    constant letter z letter e letter r letter o;
    selector letter n letter i letter l letter t
      letter a letter b letter l letter e
      field table letter n letter i letter l letter t
      letter a letter b letter l letter e;

              etc. see AM 5.2.1. to 5.2.5.

NEST standard declaration of DEC:
    where ( DEC ) is ( STANDARD ),
    EMPTY.
```

There are two kinds of 'external's: 'user external's and 'standard external's. 'User external's (AM 5.1) require a declaration in the program. 'Standard external's, however, appear in the NEST without explicit declaration. This is effected by allowing the declaration to produce EMPTY if DEC is a production of STANDARD.

Note that the NEST does not contain any indication about 'user', 'external' or 'standard'. The decision is made when the declaration is produced.

6. Pragmats.

```
NEST pragmat:
    pragmat symbol,
    NEST pragmat item list,
    point symbol.
NEST pragmat item list:
    NEST pragmat item,
    NEST pragmat item list tail option.
NEST pragmat item list tail:
    comma symbol,
    NEST pragmat item list.
NEST pragmat item:
    NEST pragmat TAG identifier,
    NEST pragmat item tail option.
NEST pragmat item tail:
    equals symbol,
    NEST pragmat definition.
NEST pragmat definition:
    integral denotation;
    string denotation;
    NEST pragmat item;
    NEST pragmat item list pack.
NEST pragmat TAG identifier:
    TAG.
```

Although the NEST is not used in any of these rules, it is passed on to facilitate the insertions of rules which implement certain pragmats.

7. Identifiers.

All identifiers are produced through rules ending in 'applied identifier' or 'defining identifier', and checking is concentrated in these.

7.1. Defining identifiers.

```
NEST1 DESCR TAG NEST2 DESCR TAG defining identifier:
    where TAG not in NEST1 NEST2,
    TAG.
FDESCR TAG defining identifier:
    TAG.
local TAG defining identifier:
    TAG.
where TAG not in NEST DESCR TAG1:
    unless ( TAG ) is ( TAG1 ),
    where TAG not in NEST.
where TAG not in EMPTY:
    where true.
```

7.2. Applied identifiers.

```
FITAB:: file; FIELDS table.
FITACK:: FITAB; FIELDS stack.

NEST LF FITAB TAG applied identifier:
    LF formal FITAB TAG applied identifier;
    where TAG not in LF,
    NEST global FITAB TAG applied identifier.
NEST LF FIELDS stack TAG applied identifier:
    LF formal FIELDS stack TAG applied identifier;
    where TAG not in LF,
    NEST global DEPENDSETY FIELDS stack TAG applied identifier.
NEST1 DESCR TAG NEST2 global DESCR TAG applied identifier:
    TAG.
new FORMALSETY1 FITACK TAG FORMALSETY2 BLOCKSETY formal
  FITACK TAG applied identifier:
    TAG.
LF BLOCKSETY1 COMPOUND TYPE EFFECT label TAG LOCALSETY BLOCKSETY2
  label TYPE EFFECT label TAG applied identifier:
    TAG.
NOTION1 FORLOC INIT TAG NOTION2
  FORLOC INIT TAG applied identifier:
    TAG.
```

8. Alphabetic listing of metaproduction rules.

```
ACCESS:: accessible; inaccessible.
AFFIX:: actual affix; destination.
ALPHA:: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q; r;
    s; t; u; v; w; x; y; z.
BLOCK:: COMPOUND LABELETY LOCALSETY.
BLOCKS:: BLOCK; BLOCKS BLOCK.
BLOCKSETY:: BLOCKS; EMPTY.
CAL:: i; CAL i.
CALETY:: CAL; EMPTY.
COMPOUND:: ACCESS compound.
CONSTABLE:: CONSTANT; VARIABLE.
CONSTAG:: CONSTANT TAG.
CONSTAGS:: CONSTAG; CONSTAGS CONSTAG.
CONSTAGSETY:: CONSTAGS; EMPTY.
CONSTANT:: DEPENDSETY constant.
DEC:: DESCR TAG.
DECS:: DEC; DECS DEC.
DECSETY:: DECS; EMPTY.
DEF:: TACK; constant; variable; rule; file; EXTERN.
DEPEND:: dependent on TAG.
DEPENDS:: DEPEND; DEPENDS DEPEND.
DEPENDSETY:: DEPENDS; EMPTY.
DESCR:: STACK; NONSTACK.
DIGIT:: digit NUMERIC.
EFFECT:: side; nonside.
EMPTY:: .
EXTERN:: external constant; external table;
    external TYPE EFFECT rule; standard.
FDESCR:: formal FITACK; formal INIT.
FIELD:: SELECTORS field.
FIELDS:: FIELD; FIELDS FIELD.
FIELDSETY:: FIELDS; EMPTY.
FILE:: file.
FITAB:: file; FIELDS table.
FITACK:: FITAB; FIELDS stack.
FORLOC:: formal; local.
FORM:: formal uninit TAG.
FORMAL:: FDESCR TAG.
FORMALS:: FORMAL; FORMALS FORMAL.
FORMALSETY:: FORMALS; EMPTY.
FORMREQ:: req form FORMSETY req loc EMPTY.
FORMS:: FORM; FORMS FORM.
FORMSETY:: FORMS; EMPTY.
INDIR:: index; direct.
INIT:: init; halfinit; uninit.
INOUT:: in; inout; out.
LABEL:: TYPE EFFECT label TAG.
LABELETY:: LABEL; EMPTY.
LASTMEMBER:: MEMBER; TERMINATOR.
LETTER:: letter ALPHA.
```

```
LF:: new FORMALSETY; LF BLOCK.
LIMIT:: MICA limit.
LOC:: local uninit TAG.
LOCAL:: local INIT TAG.
LOCALS:: LOCAL; LOCALS LOCAL.
LOCALSETY:: LOCALS; EMPTY.
LOCS:: LOC; LOCS LOC.
LOCSETY:: LOCS; EMPTY.
MEMBER:: affix form; compound member; OPERATION.
MEMBIX:: member; affix.
MICA:: max; min; calibre.
NEST:: DECSETY.
NONSTACK:: CONSTABLE; TABLE; FILE; RULE.
NOREQ:: req form EMPTY req loc EMPTY.
NOTETY:: NOTION; EMPTY.
NOTION:: ALPHA; NOTION ALPHA.
NUMERIC:: zero; one; two; three; four; five; six; seven;
    eight; nine.
OBJECT:: FIELD; LOC; FORM.
OBJECTS:: FIELDS; LOCS; FORMS.
OBJECTSETY:: OBJECTS; EMPTY.
OPERATION:: transport; identity; extension.
PARAM:: INOUT parameter; file parameter; TACK CAL parameter.
PARAMS:: PARAM; PARAMS PARAM.
PARAMSETY:: PARAMS; EMPTY.
PLAIN:: integral; character.
REQ:: req form FORMSETY req loc LOCSETY.
RESTRICTED:: free; DEPENDSETY.
RULE:: TYPE EFFECT with PARAMSETY rule.
RULEBODY:: alternative series; classification.
SELECTOR:: selector TAG.
SELECTORS:: SELECTOR; SELECTORS SELECTOR.
SELECTORSETY:: SELECTORS; EMPTY.
SIZE:: relative; absolute; absent.
STABLE:: STACK; TABLE.
STACK:: DEPENDSETY FIELDS stack.
STANDARD:: infallible nonside with in parameter in parameter
      out parameter out parameter rule
      letter a letter d letter d;
    fallible nonside with in parameter in parameter rule
      letter l letter e letter s letter s;
    infallible side with
      in parameter in parameter out parameter rule
      letter r letter a letter n letter d letter o letter m;
    fallible side with file parameter stack i parameter
      out parameter rule letter g letter e letter t letter l
      letter i letter n letter e;
    constant letter z letter e letter r letter o;
    selector letter n letter i letter l letter t
      letter a letter b letter l letter e
      field table letter n letter i letter l letter t
      letter a letter b letter l letter e;
```

etc. see AM 5.2.1. to 5.2.5.

```
TABLE:: FIELDS table.
TACK:: table; stack.
TAG:: LETTER; TAG LETTER; TAG DIGIT.
TEF:: TYPE; EFFECT.
TERMINATOR:: failure; success; exit; jump.
THING:: NOTION; ( NOTETY1 ) NOTETY2; THING ( NOTETY1 ) NOTETY2.
TYPE:: fallible; infallible.
VARIABLE:: variable.
WHETHER:: where; unless.
```

Table of Contents.