stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 91/77 NOVEMBER
(DEPARTMENT OF COMPUTER SCIENCE)

A.P.W. BÖHM

ALICE: AN EXERCISE IN PROGRAM PORTABILITY

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): 68A00,68A05

Comp. Review: 4.12 4.21 4.22

ALICE: An Exercise in Program Portability

by

A.P.W. Böhm

ABSTRACT

ALEPH has been designed at the Mathematical Centre as a tool for compiler writing. In the ALEPH compiler a clear interface between the machine-independent and the machine-dependent part is established. This interface presents itself as an intermediate code, called ALICE. ALICE can be implemented easily and efficiently on many machines. This report defines ALICE, gives installation hints, and states which design decisions were taken.

KEY WORDS & PHRASES: ALEPH, abstract machine, intermediate code, portability, adaptability, machine-independence

INDEX

0 Preface	4
0.1 Informal and incomplete introduction to ALEPH	5
1 The role of ALICE in the compilation of ALEPH programs	8
1.1 Splitting up the ALEPH compiler	8
1.2 Porting an ALEPH program to a new machine	9
2 Outline of ALICE	12
2.1 ALICE grammars	12
2.2 The format of ALICE statements	13
2.3 Features needed to translate ALICE	14
2.4 The character set	16
2.5 The simple ALICE grammar	18
2.6 Example	27
3 Description of the various constructs of ALICE	29
3.1 Values	29
3.1.1 The range of integer values	30
3.1.2 The character code	32
3.1.3 The address space	33
3.1.4 Strings	34
3.1.5 Example	36
3.2 Data	39
3.2.1 Integer	39
3.2.1.1 Constant source	39
3.2.1.2 Variable	42
3.2.2 List	43
3.2.2.1 List area	43
3.2.2.2 List administration	45
3.2.2.3 External table	47
3.2.3 File	48
3.2.3.1 File administration	48

3.3 Rules 51
3.3.1 Introduction 51
3.3.2 The ALICE abstract machine 53
3.3.3 Outline of an ALICE rule 55
3.3.4 Rule head 55
3.3.5 Rule tail 57
3.3.6 Calling a rule 60
3.3.7 Parameter passing 62
3.3.7.1 Actual input parameters 62
3.3.7.2 Actual output parameters 67
3.3.8 Primitive rules 71
3.3.8.1 External rules 71
3.3.8.2 ALICE primitives 73
3.4 Program structure 78
3.5 ALICE grammar 81

4 References 99

0 Preface

At the moment this report is written, we have an ALEPH compiler running happily on a CYBER under SCOPE 3.4.1. This ALEPH compiler generates COMPASS, the assembly language of the CYBER.

ALEPH was designed by people working on the ALGOL 68 compiler at the Mathematical Centre. The most important task of the ALEPH compiler is to compile this MC ALGOL 68 compiler, which is written in ALEPH. The ALGOL 68 group is trying to develop a both portable and adaptable compiler for the full language ALGOL 68.

A necessary condition for reaching this honourable goal is to have a portable and adaptable ALEPH compiler as well. The present ALEPH compiler does not meet this end, and therefore it is being rewritten to establish a clear interface between its machine-independent and its machine-dependent part.

Roughly speaking, the machine-independent part of the new compiler consists of a parser and a semantic analyzer; the machine-dependent part consists of a code generator. The interface between these two parts presents itself as an intermediate code. This intermediate code, called ALICE, can be considered the ultimate machine independent stage in the translation of ALEPH to machine code.

In a first version of the ALEPH compiler on a new machine, ALICE is the output of the semantic analyzer, that is, it is written on a file and read by a translator from ALICE to machine code. In a more progressed version ALICE is merely a set of calls to code-generating routines. More precisely, the semantic analyzer ends with calls to code generating routines, which in the first case simply copy the call to a file and in the second case generate code for a particular machine.

Although this report is primarily meant for installers of the ALEPH compiler, who must be familiar with ALEPH [1], it is self-contained. A rather informal and incomplete introduction to ALEPH is given below. Chapter one of this report describes the role of ALICE in the compilation process from ALEPH to machine code in detail. Chapter two describes the format of the ALICE statements and the character set in which ALICE programs are written. Chapter two also mentions some programming tools which can be used to to facilitate the translation from ALICE to machine code. Chapter three contains a complete description of the various constructions in ALICE.

0.1 Informal and incomplete introduction to ALEPH

For those not intending to implement ALEPH and unfamiliar with it, a very informal and incomplete introduction into its use now follows.

ALEPH is a simple programming language, based on two features:

1. Decomposition by means of a procedure mechanism. Procedures are called "rules" in ALEPH.

2. Specification of alternatives within a rule.

ALEPH originated from the idea that a context-free grammar can be used as a programming language, such as in the recursive descent parsing method. For instance, the grammar for strict binary trees in parenthesized infix notation:

```
tree                 : terminal node;
                       open symbol, tree, non terminal node, tree,
                            close symbol.

terminal node        : letter.

non terminal node    : letter.
```

can be considered to have the following meaning:

```
there is a tree:
either if there is a terminal node
    or if there is  an open-symbol,  followed by  a tree,  followed by a
                    non-terminal-node, a tree, and a close-symbol.
```

An ALEPH program is a set of "rule declarations" and "data declarations". Just as in a context-free grammar, there is one starting point called the "root" of the program. The root calls one rule, which calls other rules, and so the ALEPH machine comes into action. A rule consists of a number of "alternatives". An alternative consists of a number of "members". A member is either a language primitive, such as an assignment, or a call to another rule.

The first member of an alternative can be considered as the key to that alternative: if the key fits the other alternatives are no longer of interest. To be able to serve as keys, members can fail or succeed. A member succeeds if the rule it calls or the language primitive it represents succeeds. A rule succeeds if one of its alternatives applies and all members of that alternative succeed. A rule fails either if no alternative applies or if an alternative applies but one of its members fails. Some rules do not have to be declared, but can be considered as built in. Their names are known to the ALEPH compiler. These rules are called external rules. The recursive definition of success or failure of a member does not loop because of the existence of the external rules and the language primitives.

For some rules in a program there will always be an alternative that applies: these rules cannot fail. Therefore rules can be divided in two groups: those that can fail and those that cannot.

There is another criterion that divides rules in two groups: rules have or have no "side effects". A rule has side effects if it changes the environment (global data).

The division criteria can be combined, yielding four groups of rules:

can fail, has side effects	: 'predicate'
can fail, has no side effects	: 'question'
cannot fail, has side effects	: 'action'
cannot fail, has no side effects	: 'function'

This information about a rule is given by the programmer and checked by the compiler.

Communication between the members in a rule and between callers and called rules is done by parameters, called "affixes". Besides "formal" affixes a rule has "local" affixes for scratch pad purposes. Formal affixes are either prefilled at call entry (input parameters), or used by the caller upon return (output parameters), or both (input-output parameters). Local affixes are uninitialized at call entry. If the rule fails, the caller will not need the values of the output affixes: they will not even be passed back at call exit.

As an example, part of a program that reads a sequence of numbers and prints their sum is showed.

```
'root' read and print.

'action' read and print - sum:

        read + sum, print + sum.

'action' read + res>:

        number + res, rest numbers option + res.


'action' rest numbers option + >res> - nmb:

        comma symbol, number + nmb, plus + nmb + res + res,
                rest numbers option + res;
        +.


'action' number + res>:

        get int + input file + res;
        error + bad number, 0 -> res.
```

```
'action' print + >number:

        put int + output file + number.
```

The pluses connect the affixes to the rules. The minus signals a local affix. The right arrow in front of "res" in "rest numbers option" indicates that "res" is an input affix. The value of the actual affix of the caller will be copied to the formal affix of the called rule. The right arrow at the back of "res" indicates that at rule exit its value will be restored to the actual affix of the caller.

The local affix "sum" in "read and print" is uninitialized at the colon. From the declaration of "read" it follows that it will return a value to "sum". After the call of "read", "sum" is initialized and can be used by "print".

"get int" is an external predicate. The programmer is forced to think about the exceptional case end of file. "0 -> res" assigns 0 to "res", as might have been expected.

Data types.

ALEPH restricts itself to integer data and stacks of these. Global variables must be initialized upon declaration. Stacks have the usual property that top elements may be added, inspected and removed. In addition, they have the following properties:
a. All elements can be reached; thus the stack can act as an array.
b. Bottom elements can be removed; thus the stack can act as a queue.

1 The role of ALICE in the compilation of ALEPH programs

1.1 Splitting up the ALEPH compiler

The aim is an ALEPH compiler that can run on a variety of machines (portable) and that can generate code for a variety of machines (adaptable).

It is clear that the code generator of a compiler belongs to the machine dependent part of that compiler, provided that all optimizations that can be done machine independently are not considered as part of the code generator. In most compilers, however, the code generator is not the only part that contains machine dependencies. The first ALEPH compiler, for instance, had this regrettable quality. In the new version of the ALEPH compiler all machine dependencies are located in the code generator. This is necessary to adapt the ALEPH compiler easily to generate code for a new machine.

The machine dependencies should not only be grouped together in one module, but this module must be easily separable from the rest of the compiler as well. This implies that a clear interface between the machine-independent and the machine-dependent part is another necessity for portability and adaptability.

The ALEPH compiler consists of the following parts:

lexical analyzer and parser,
semantic analyzer and machine-independent code generator,
machine-dependent code generator.

The lexical scanner and the parser make up the first scan, while the semantic analyzer and the machine-independent code generator form the second scan of the compiler. These two scans will not have to be changed when the compiler is moved to another machine. They are written in ALEPH. This is a vital property, and enables the machine independent part of the compiler to be moved just as any other machine independent ALEPH program. The design of the ALEPH compiler will be described elsewhere.

The output of the second scan consists of a machine-independent intermediate code called "ALICE". This code can be thought of as the assembly language of an abstract machine. The machine-dependent code generator translates ALICE to machine code and must be rewritten for every new machine. It must therefore be easy to translate ALICE to various kinds of machine codes. The format of ALICE must be simple enough to be translated by a macro-processor or macro-assembler.

The ALICE statements must deliver information to the machine-dependent code generator when the machine-dependent code generator needs that information, so that ALICE macros can be processed one by one. Unfortunately, this need varies from implementation to implementation. ALICE is therefore a rather redundant language. ALICE programs must be, on the other hand, manageable in size and not all the needs for future implementations can be predicted. It is impossible to supply each bit of information where it might be needed by an implementation. Some trade-offs are made, whether on purpose or not.

The need for the machine-dependent code generator to be easily separable from the rest of the compiler lays two obvious requirements on ALICE. The first one is that ALICE has to serve as a one-way information carrier from the second scan to the machine-dependent code generator. The second is that ALICE is the only interface between the second scan and the machine-dependent code generator.

These two requirements guarantee that no information about the machine can be used by the second scan and that the machine-dependent code generator uses no global data from the rest of the compiler. This ensures that the translation from ALEPH to ALICE will not introduce machine dependencies and that an ALICE text can be translated without further reference to the original ALEPH text.

These requirements have not been easy to fulfil. The evaluation of expressions, for instance, must be postponed until machine-dependent code generation, because some values are machine dependent quantities. This, of course, has its effect on both ALICE and the first two scans of the compiler.

The alternative, on the other hand, seems even worse. If information about the machine and the implementation can flow back from the machine-dependent code generator to the second scan, the uniqueness of the ALICE translation of an ALEPH program is lost. For every different target machine a different ALICE would have been created. This could cause a discouraging feedback loop between host machine and target machine. Furthermore, if the machine-dependent code generator were to use global data from the rest of the compiler, it would become impossible to put ALICE, the interface between them, on a file and send it to a new machine.

1.2 Porting an ALEPH program to a new machine.

Suppose that a program written in ALEPH has to be ported to a new machine which has no ALEPH compiler available. The program is translated to ALICE and written in a standard format, on some information carrier such as a magnetic tape. Together with this program, other software to make the bootstrapping as easy as possible, is ported. To pass this stage of the porting process smoothly, the character set of ALICE is kept to a bare minimum [2].

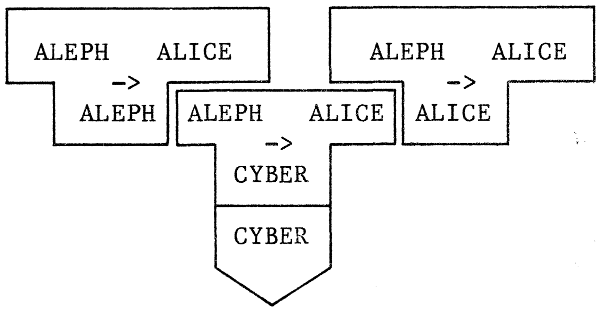
To run the ALICE program on the new machine, a translator from ALICE to machine code and a suitable run-time system have to be constructed by the installer. The installer has to make decisions about the evaluation of expressions, the data-representation, the allocation of tables and stacks, the calling mechanism and the passing of parameters. These decisions will be influenced by the architecture of the new machine, the ALEPH program(s) that have to be run, and the amount of time the installer has available.

If the machine is small or the ALEPH programs to be run are going to need large tables and stacks, the installer may have to provide a means for allocating tables and stacks on background memory. The installer may decide to generate very compact code that has to be interpreted by a small driver which is part of his run-time system, to keep the storage needed for the code as small as possible.

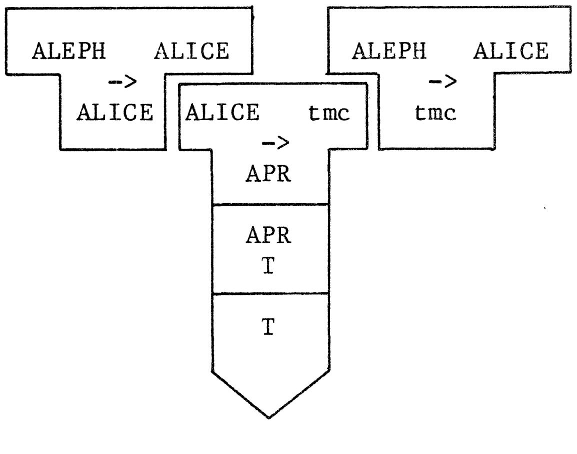
The first implementation may be inefficient but easily set up. The installer can, for instance, decide to use the ALICE to FORTRAN translator written in FORTRAN, which will be part of the software sent along on the tape. If he does this, a lot of decisions do not have to be bothered about anymore, and all that has to be done is getting a big FORTRAN program running. The code derived this way will be very inefficient only acceptable in a first bootstrap.

The first ALICE-to-machine-code translator may be realized by means of a macro-processor. This will make the translation easy as long as the installer does not want fancy optimizing, which requires reading of a number of ALICE statements together, code motion or other tricks.

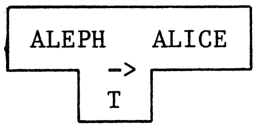
The T-diagrams below describe the various stages which the implementation of an ALEPH compiler on a new machine has to go through.



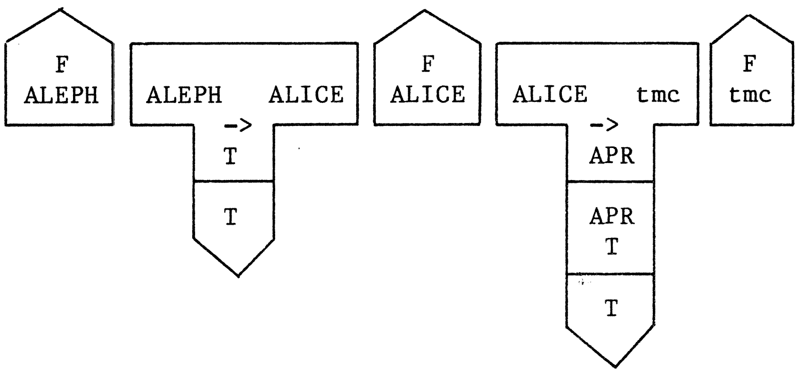
The translation of the ALEPH compiler on the host machine to get the ALICE version of it is sketched above. This ALEPH to ALICE compiler in ALICE will be sent to the target machine.



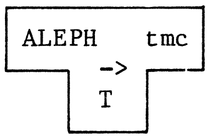
The above T-diagram exemplifies the translation of the ALEPH compiler on the target machine by an ALICE processor (APR), to the ALEPH compiler written in target machine code (tmc). APR might be a macro-processor fed with appropriate macro-definitions. Together with a suitable run-time system this will give a first version of the ALEPH compiler:



The first version of the ALEPH compiler will generate ALICE. So for every ALEPH program to be translated to machine code the ALICE processor is needed:



This process can be cut short by changing the machine-independent code generator of the ALEPH compiler (still generating ALICE) into a machine-dependent one, generating tmc. How this change has to be made is shown by the ALICE to tmc translator, which is already written and tested. This will give a final version of the ALEPH compiler on the new machine:



2 Outline of ALICE

2.1 ALICE grammars

In the sequel of this report, context-free rules are used to describe the format of ALICE statements. The fancy way to describe the format of these context-free rules is of course by writing a context-free grammar for them, but let us keep things simple. The context-free rules consist of a left hand side terminated by a colon and a right hand side terminated by a period. The alternatives of a right hand side are separated by semicolons and the notions making up an alternative are separated by commas. Notions consist of letters. Square brackets around a phrase (a piece of an alternative) are shorthand to indicate two alternatives: one with the phrase and one without it, so

```
a    : b, [c, d], e.
```

means

```
a    : b, c, d, e;
       b, e.
```

Terminal notions are those ending with "symbol", such as "end symbol", and the notions:

"character", "end of line", "string", "integer".

Terminal notions ending with "symbol" are represented by three-letter tags. A string is a sequence of characters surrounded by quotes; a quote within the string is represented by a quote-image (two quotes). An integer is an unsigned sequence of decimal digits. Comments in the grammar are surrounded by dollar tokens.

There are two context-free grammars describing ALICE. The first one is a very simple grammar, which defines ALICE as an almost unstructured sequence of macros. This grammar does not define the order in which the macros will occur and the language produced by this grammar is therefore a superset of ALICE. It still is a useful description of ALICE because it may serve as a guide for the writing of a set of macro-definitions for a macro-processor that is, in most cases, going to be the first translator from ALICE to target machine code. In other words, there is a one-to-one correspondence between the macros of the first grammar and the macro-definitions in the macro-processor version of the ALICE translator.

The second grammar is more complicated, because it describes the structure of ALICE and the order in which the macros can occur. This grammar is written to define ALICE precisely and to make the meaning of the various macros understandable. Together with the other sub-sections of section 3, this grammar is the ultimate description of ALICE.

2.2 The format of ALICE macros

ALICE statements are sometimes called macros in this report, even though processors other than macro-processors may be used to translate ALICE. An ALICE statement consists of a macro name possibly followed by a space and a number of parameters separated by commas. One macro is written per line, or card or record. The macros are delimited in a medium dependent way. In the grammar it says that macros are delimited by ´end of line´s, but for every system this may mean something different.

Examples:

```
jmp 15
lab 15
str "Ewige Blumenkraft"
erl gtc
efc 10,gtc,2,0,15
chd 1,a
```

This format has been chosen because it is easily adapted to the format demanded by most macro-assemblers. It is inevitable that some assemblers will already use macro names of ALICE. Solutions for these problems must depend on the local circumstances. One could, for instance, decide to prefix all ALICE macros with some special character or to change some of the macro names of ALICE by means of an editor.

General-purpose macro-processors, such as ML/1 and STAGE2, also find this format easy to digest. In ML/1 [3] a macro can be used to describe the delimiter structure of the parameter part of all macros (the macro called "params" in section 2.6). This macro definition makes the other macro definitions look less sloppy. ML/1 does not consider ´end of line´ as ´end of macro´. It is therefore possible to group ALICE macros in one macro definition in ML/1. In STAGE2 [4] every macro is supposed to be written on one line. There is no problem with the absence of a special ´end of macro´ character, since STAGE2 accepts ´end of line´ for this.

The first ALICE-to-pdp11/45-assembly-language translator was written in ML/1 without problems. This was done while ALICE was still changing frequently. ML/1 was chosen, first, because it was available and second, because it is easy to change the delimiter structures of macro definitions in ML/1. Once ALICE had been stabilized, an ALICE-to-assembly-language translator was written in a high-level language, because large ML/1 programs are hardly readable, let alone self documenting and, because ML/1 programs run slowly and demand lots of storage.

2.3 Features needed to translate ALICE

It is not enough for the ALICE translator to process single macros independently, because some of the macros communicate with each other. The translator has to keep track of this communication. In all cases but one, a fixed number of macro-time global variables (a macro-time variable is a variable of the ALICE translator) can serve this purpose. The exception is the communication between evaluating expressions, declaring constants and referring to them in executable code. The feature needed to handle this case, a macro-time symbol table, will be discussed now.

Roughly speaking, an ALICE program consists of three parts:

1: values
2: data
3: rules

In the ´values´ part all expressions are evaluated and information to administrate lists and files is gathered. Both expression evaluation and the gathering of this information can be done at macro time (when ALICE is translated) and therefore no assembly code has to be generated for it. The ALICE processor has to store the values and use them later on when macros belonging to the ´data´ and ´rules´ part are translated.

The ´data´ part declares all constant sources, variables, lists, initial list contents (called list filling), and the administration structures for lists and files. Constant sources are constant actual parameters of calls.

Example:

Consider the following piece of an ALEPH program:

```
´constant´ char distance = /1/ - /0/,
           character 0 = /0/.

´action´ print digit + >digit - int - char:
           times + digit + char distance + int,
           add + int + character 0 + char,
           print char + char.
```

In the ´values´ part of the ALICE equivalent of this program the expressions "/1/ - /0/" and "/0/" will be evaluated:

```
chd 1,1
chd 2,0
sub 3,1,2
```

The first two macros are character-denotation macros. They demand a conversion from character to integer of their second parameter. The value thus obtained will be referred to by their first parameter. The third macro demands a subtraction of the two values just obtained, the result of which will be referred to by its first parameter.

In the ′data′ part two constant sources corresponding to "char distance" and "letter 0" will be declared. These constant sources have values calculated in the ′values′ part. The first parameter of these macros stands for the name of the datum, while the second refers back to values:

```
css 1,2
css 2,3
```

In the ′rules′ part the ALICE code for the constant actual parameter of the affix form "times + digit + char distance + int" will consist of a macro referring back to both the ′data′ macro "css 2,3" and the ′values′ macro "sub 3,1,2" involved:

```
lvc ---,2,3,---
```

One way to translate these macros is to build up a symbol table while processing the ′value′ macros, to generate assembly-language data declarations for the ′data′ macros, and to refer to these declarations in the code generated for the ′rules′ macros. In this scheme the ′rules′ macros cause no problems because the parameters give all information that is needed. How to handle the communication between the ′value′ and ′data′ macros will be shown below for both ML/1 and STAGE2.

In ML/1 dynamically defined macros have to serve for building the symbol table. Character-to-integer conversion is not a primitive available in ML/1, so this has to be done by means of macro definitions, such as:

```
mcdef chtoint0 as 48
mcdef chtoint1 as 49
```

STAGE2 offers exactly the primitives needed for character conversion and symbol table manipulation. For the latter the associative addressing concept of STAGE2 can be used.

An example of an ML/1 version and a STAGE2 version of the macro-definitions for the macros with macro-names "chd", "int", "sub" and "css" is given in section 2.6.

2.4 The character set

Suppose a program written in ALEPH, say an ALGOL 68 compiler, has to be ported to a new machine and that this program contains all sorts of weird characters like broken backslashes and so on. How should this program be ported to the new machine?

Not in ALICE but in ALEPH! (section 1.2)

For one thing, when an ALEPH program is translated to ALICE it tends to become very big and almost unreadable for human beings. So if one can avoid porting programs in ALICE, one had better avoid it.

There is one program that must be ported in ALICE: the ALEPH compiler. Apart from the compiler and test programs for the ALICE system, all ALEPH programs to be ported will be ported in ALEPH.

The first thing to do is to implement the ALEPH compiler on the new machine. The ALEPH compiler doesn′t contain characters other than the ′worthy′ ALEPH characters and newlines. The fifty-six worthy ALEPH characters are:

```
A to Z
0 to 9
space
+ * , . / " ′ ( ) [ ] - : ; = < > $ #
```

This set of worthy ALEPH characters is a proper subset of the worthy ALGOL 68 characters [5], and also a subset of ASCII and EBCDIC. ′end of line′ is no worthy character because on some installations it isn′t even a character.

If ALEPH programs are to be compiled on the new machine this set of worthy ALEPH characters is the minimal set needed.

If the new machine can′t provide this minimal set or a reasonable equivalent, special machine dependent arrangements are needed, because the ALICE version of the compiler contains the characters of the minimal set and also, because some character code has to be chosen for representing the worthy characters in the ALEPH programs that will have to be compiled in a later stage, anyway. These problems are very machine dependent and can only be solved at the new machine.

The character set needed for ALICE is identical to the character set needed for ALEPH. Therefore the ALICE version of the ALEPH compiler does not give rise to additional problems as far as the character set is concerned.

Now let us suppose that the ALEPH compiler has been installed and the compiler still generates ALICE. ALICE plays a different role now: it serves as an interface between two parts of the compiler, running on the same machine. ALICE might only have survived as a set of calls from the code-generating routines that have become part of the second scan.

At this stage the ALGOL 68 compiler, written in ALEPH, can be ported. If the weird characters belong to the character set available on the new machine, they can be transferred via ALICE in exactly the same way the worthy ALEPH characters will be transferred. If they cannot be read in, some escape mechanism has to be designed for them.

But this is not a problem on the ALICE level, so in ALICE we shall not bother about it. It does not help to distribute the problem of the weird characters over all phases of the compilation process and it is certainly impossible to solve this problem for all character sets on all machines.

In the sequel we shall assume that the only characters in ALICE programs that are to be ported to new machines are the worthy ALEPH characters and that the characters in the ALICE interface between two parts of a compiler on one machine are the characters available on that machine.

2.5 The simple ALICE grammar

```
$Comments such as this one are surrounded by dollar tokens$


$ALICE terminal symbols         representation   $



$macro names$
add symbol;                     $       add      $
begin file adm symbol;          $       bfa      $
call id symbol;                 $       cll      $
class begin symbol;             $       csb      $
class end symbol;               $       cse      $
char denotation symbol;         $       chd      $
constant source symbol;         $       css      $
communication symbol;           $       cmm      $
copy a reg symbol;              $       car      $
copy from input gate symbol;    $       cig      $
copy v reg symbol;              $       cvr      $
divide symbol;                  $       dvd      $
end file adm symbol;            $       efa      $
end list symbol;                $       els      $
end symbol;                     $       end      $
end values symbol;              $       eva      $
exit symbol;                    $       ext      $
ext fcall symbol;               $       efc      $
ext scall symbol;               $       esc      $
ext table length symbol;        $       etl      $
ext table decl symbol;          $       etd      $
extcall end symbol;             $       ece      $
ext scall id symbol;            $       esi      $
ext fcall id symbol;            $       efi      $
extension call symbol;          $       etc      $
extension copy symbol;          $       exc      $
extension end symbol;           $       exe      $
extension id symbol;            $       exi      $
extrule decl symbol;            $       erl      $
fail tail id symbol;            $       fti      $
fallow symbol;                  $       flw      $
fcall symbol;                   $       fcl      $
class box id symbol;            $       cbi      $
end class box symbol;           $       ebx      $
free w reg symbol;              $       frw      $
indexed input parameter symbol; $       iip      $
indexed output parameter symbol;$       iop      $
input gate symbol;              $       igt      $
int symbol;                     $       int      $
int fill symbol;                $       itf      $
jump symbol;                    $       jmp      $
label symbol;                   $       lab      $
link symbol;                    $       lnk      $
```

```
list adm symbol;                  $       ldm      $
list symbol;                      $       lst      $
loada global symbol;              $       lag      $
loada stack var symbol;           $       las      $
loadv constant symbol;            $       lvc      $
loadv limit symbol;               $       lvl      $
loadv list elem symbol;           $       lvi      $
loadv stack var symbol;           $       lvs      $
loadv variable symbol;            $       lvv      $
loadw symbol;                     $       ldw      $
manifest constant symbol;         $       mcn      $
multiply symbol;                  $       mul      $
rule id symbol;                   $       rli      $
numerical symbol;                 $       num      $
output gate symbol;               $       ogt      $
pointer symbol;                   $       ptr      $
program id symbol;                $       pid      $
restore to output gate symbol;    $       rog      $
root symbol;                      $       rut      $
source line symbol;               $       srl      $
scall symbol;                     $       scl      $
stack frame symbol;               $       sfr      $
status symbol;                    $       sts      $
storew variable symbol;           $       swv      $
storew list element symbol;       $       swi      $
storew stack var symbol;          $       sws      $
string length symbol;             $       sln      $
string fill symbol;               $       str      $
subtract symbol;                  $       sub      $
success tail id symbol;           $       sti      $
target stack frame symbol;        $       tsf      $
unstack and return symbol;        $       unr      $
variable symbol;                  $       var      $
zone bounds symbol;               $       znb      $
zone value symbol;                $       znv      $

$delimeters$
space symbol;                     $       " "      $
comma symbol;                     $        ,       $
end of line;                      $medium dependent$

$parameters$
new line symbol;                  $       nln      $
same line symbol;                 $       sln      $
rest line symbol;                 $       rln      $
new page symbol;                  $       npg      $
max char symbol;                  $       mxc      $
word size symbol;                 $       wsz      $
max int symbol;                   $       mxi      $
min int symbol;                   $       mni      $
int size symbol;                  $       isz      $
comma-tag symbol;                 $       com      $
space-tag symbol;                 $       spc      $
min addr symbol;                  $       mna      $
max addr symbol;                  $       mxa      $
virt length symbol;               $       vln      $
```

```
nil symbol;                     $       nil     $
false symbol;                   $       fls     $
true symbol;                    $       tru     $

add-tag symbol;                 $       add     $
subtr symbol;                   $       sub     $
mult symbol;                    $       mul     $
divrem symbol;                  $       div     $
plus symbol;                    $       pls     $
minus symbol;                   $       min     $
times symbol;                   $       tms     $
incr symbol;                    $       inc     $
decr symbol;                    $       dec     $
less symbol;                    $       les     $
lseq symbol;                    $       lsq     $
more symbol;                    $       mor     $
mreq symbol;                    $       mrq     $
equal symbol;                   $       eql     $
noteq symbol;                   $       ntq     $
random symbol;                  $       rnd     $
set random symbol;              $       srn     $
set real random symbol;         $       srr     $
sqrt symbol;                    $       sqr     $
pack int symbol;                $       pki     $
unpack int symbol;              $       upi     $
bool invert symbol;             $       biv     $
bool and symbol;                $       bnd     $
bool or symbol;                 $       bor     $
bool xor symbol;                $       xor     $
left circ symbol;               $       lci     $
right circ symbol;              $       rci     $
right clear symbol;             $       rcl     $
is elem symbol;                 $       isl     $
is true symbol;                 $       itr     $
is false symbol;                $       isf     $
set elem symbol;                $       stl     $
clear elem symbol;              $       cll     $
extract bits symbol;            $       exb     $
first true symbol;              $       ftr     $
pack bool symbol;               $       pkb     $
unpack bool symbol;             $       upb     $
to ascii symbol;                $       tsc     $
from ascii symbol;              $       fsc     $
pack string symbol;             $       pks     $
unpack string symbol;           $       ups     $
string elem symbol;             $       ste     $
string length symbol;           $       stl     $
compare string symbol;          $       cms     $
unstack string symbol;          $       uns     $
previous string symbol;         $       pvs     $
was symbol;                     $       was     $
next symbol;                    $       nxt     $
previous symbol;                $       prv     $
list length symbol;             $       lsl     $
unstack symbol;                 $       utk     $
unstack to symbol;              $       ust     $
```

```
unqueue symbol;                 $       unq     $
scratch symbol;                 $       scr     $
get line symbol;                $       gln     $
put line symbol;                $       pln     $
get char symbol;                $       gch     $
put char symbol;                $       pch     $
put string symbol;              $       pst     $
get int symbol;                 $       gnt     $
put int symbol;                 $       pnt     $
get data symbol;                $       gdt     $
put data symbol;                $       pdt     $

$other primitives used as parameters$
string;                         $character sequence
                                 delimited by quotes
                                 quotes in the string are represented by
                                 quote images ("")  $

character;
integer.                        $digit sequence$



ALICE program            : program id symbol, sp, string, el,
                                                  $title string$

                           status information,

                           values,
                           end values symbol, el,
                           data,

                           communication area, $ends data declarations$

                           rules,

                           end symbol, sp, string, el. $title string$


sp                       : space symbol.
co                       : comma symbol.
el                       : end of line.


status information       : status symbol, sp,
                           integer, co, $maximal stack frame$
                           integer, co, $maximal gate size$
                           integer, co, $number of locations in value table$
                           integer, co, $number of variables$
                           integer, co, $number of files$
                           integer, co, $number of breathing lists$
                           integer, co, $number of non-breathing lists$
                           integer, co, $background:
                                         0: No lists on background
                                         1: Lists on background$
```

```
                          integer, el. $dump:
                                       0: no dump
                                       1: rule dump
                                       2: global dump
                                       4: member dump$

values                  : value, [values].


value                   : int symbol, sp, location, co, integer, el;
                          manifest constant symbol, sp,
                          location, co, manco, el;
                          char denotation symbol, sp, location, co,
                          character, el;
                          string length symbol, sp, location,
                          co, integer, el;
                          ext table length symbol, sp, location,
                          co, string, el;
                          operator, sp, location, co,
                          valref, co, valref, el.

manco                   : new line symbol;
                          same line symbol;
                          rest line symbol;
                          new page symbol;
                          max char symbol;
                          word size symbol;
                          max int symbol;
                          min int symbol;
                          int size symbol;
                          comma-tag symbol;
                          space-tag symbol;
                          min addr symbol;
                          max addr symbol;
                          virt length symbol;
                          nil symbol.

operator                : add symbol;
                          subtract symbol;
                          multiply symbol;
                          divide symbol.

location                : integer.
valref                  : integer.
repr                    : integer.



data                    : data item, [data].
data item               : constant source symbol, sp, repr,
                          co, valref, el;
                          list symbol, sp, repr, co, list type, co,
                          valref, el;
                          end list symbol, sp, repr, co,
                          list type, co, valref, el;
                          int fill symbol, sp, valref, el;
                          string fill symbol, sp, string, el;
                          fallow symbol, sp, valref, el;
                          variable symbol, sp, repr, co, valref, co,
                          repr, co, string, el;
                          begin file adm symbol, sp, file info, el;
                          end file adm symbol, sp, file info, el;
                          numerical symbol, sp, valref, co, valref, el;
                          pointer symbol, sp, repr, el;
                          list adm symbol, sp, list info, el;
                          external table decl symbol, sp,
                          list info, co, string, el.

list info               : repr, co, integer, co, valref, co,
                          valref, co, valref, co, valref, co,
                          valref, co, repr, co, string.


file info                 repr, co, integer, co, integer, co,
                          repr, co, string, el.


communication area      : communication symbol, sp,
                          repr, co, repr, co, repr, co, string, el,
                          status information.

rules                   : rule, [rules];
                          call, [rules];
                          primitive, [rules];
                          parameter, [rules].


rule                    : extrule decl symbol, sp, repr, co, extag, el;
                          root symbol, sp, string, el;
                          rule id symbol, sp, repr, co, integer, co
                          integer, co, string, el;
                          stack frame symbol, sp, integer,
                          co, integer, el;
                          success tail id symbol, sp, repr, co,
                          integer, co, integer, el;
                          output gate symbol, sp, integer, el;
                          unstack and return symbol, sp,
                          integer, co, integer, co, return type, el;
                          fail tail id symbol, sp, repr, co,
                          integer, co, integer, el;
                          copy from input gate symbol, sp, formal, el;
                          restore to output gate symbol, sp, formal, el.


return type             : true symbol;
                          false symbol.


extag                   : add-tag symbol;
                          subtr symbol;
                          mult symbol;
                          divrem symbol;
```

```
plus symbol;
minus symbol;
times symbol;
incr symbol;
decr symbol;
less symbol;
lseq symbol;
more symbol;
mreq symbol;
equal symbol;
noteq symbol;
random symbol;
set random symbol;
set real random symbol;
sqrt symbol;
pack int symbol;
unpack int symbol;
bool invert symbol;
bool and symbol;
bool or symbol;
bool xor symbol;
left circ symbol;
right circ symbol;
right clear symbol;
is elem symbol;
is true symbol;
is false symbol;
set elem symbol;
clear elem symbol;
extract bits symbol;
first true symbol;
pack bool symbol;
unpack bool symbol;
to ascii symbol;
from ascii symbol;
pack string symbol;
unpack string symbol;
string elem symbol;
string length symbol;
compare string symbol;
unstack string symbol;
previous string symbol;
was symbol;
next symbol;
previous symbol;
list length symbol;
unstack symbol;
unstack to symbol;
unqueue symbol;
scratch symbol;
get line symbol;
put line symbol;
get char symbol;
put char symbol;
put string symbol;
get int symbol;
```

```
                         put int symbol;
                         get data symbol;
                         put data symbol.


call                   : call id symbol, sp, repr, co, integer, el;
                         input gate symbol, sp, integer, el;
                         target stack frame symbol, sp, integer, co,
                         integer, el;
                         scall symbol, sp, repr, el;
                         fcall symbol, sp, repr, co, repr, el;
                         link symbol, sp, repr, el;
                         ext scall id symbol, sp, repr, co, extag, el;
                         ext fcall id symbol, sp,
                         repr, co, extag, co, repr, el;
                         ext scall symbol, sp, repr, co, stag, el;
                         ext fcall symbol, sp,
                         repr, co, extag, co, repr, el;
                         extcall end symbol, co, repr, el.


primitive              : jump symbol, sp, repr, el;
                         source line symbol, sp, integer, el;
                         class box id symbol, el;
                         class box end symbol, sp, repr, el;
                         class begin symbol, sp, repr, co,
                         optimize, el;
                         class end symbol, sp, optimize, el;
                         zone bounds symbol, sp,
                         minbound, co, maxbound, co, repr, el;
                         zone value symbol, sp, repr, co, valref, co,
                         repr, el;
                         extension id symbol, el;
                         extension call symbol, el;
                         extension copy symbol, sp, formal, el;
                         extension end symbol, sp, repr, el;
                         exit symbol, sp, repr, co, valref, el;
                         label symbol, sp, repr, el.


minbound               : repr, co, valref.
maxbound               : repr, co, valref.
optimize               : true symbol;
                         false symbol.


parameter              : copy v reg symbol, sp, formal, el;
                         copy a reg symbol, sp, formal, el;
                         loadv constant symbol, sp,
                         repr, co, valref, el;
                         loadv variable symbol, sp, repr, el;
                         indexed input parameter symbol, el;
                         loadv list elem symbol, sp, integer, el;
loadv limit symbol, sp, integer, el;
loadv stack var symbol, sp, integer, el;
loada global symbol, sp, repr, el;
loada stack var symbol, sp, integer, el;
loadw symbol, sp, formal, el;
storew variable symbol, sp, repr, el;
indexed output parameter symbol, el;
storew list element symbol, sp, select, el;
storew stack var symbol, sp, integer, el;
free w reg symbol, el.
```

2.6 Example: ML/1 and STAGE2 symbol table usage

In this example the macro-definitions for the macros with macro names "chd", "int", "sub" and "css" (which need a symbol table to communicate with each other) are given in both ML/1 and STAGE2. In the ML/1 version, a special macro "eval(   )" is needed to evaluate pieces of text which are not a parameter of a macro call.

```
mcins !.
mcskip mt,[]
mcdef params
as [nl opt , nl or nl all]
mcdef chtint0 as 48
mcdef chtint1 as 49
mcdef chd params
as [mcdefg sym!a1. as chtint!a2.
]
mcdef int params
as [mcdefg sym!a1. as !a2.
]
mcdef sub params
as [mcset t1 = eval( sym!a2. ) - eval( sym!a3. )
mcdefg sym!a1. as !t1.
]
mcdef eval with ( )
as [mcdef temp as !a1.
temp]
mcdef css params
as [c!a1.: eval( sym!a2. )
]
```

If we feed ML/1 with these macro definitions and the following ALICE macros:

```
chd 1,1
chd 2,0
sub 3,1,2
css 1,2
css 2,3
```

we get the following result:

```
c1: 48
c2: 1
```

The STAGE2 version of the same macro definitions is:

```
.$$'0 (+-*/)
end $.
'f0$
$
$ .
'10$
$
chd $,$.
under x'10 store '28$
$
sub $,$,$.
under x'10 store eval x'20-x'30$
$
int $,$.
under x'10 store '20$
$
under $ store $.
'f3$
$
under $ store eval $.
under '10 store '24$
$
css $,$.
xcss '10,x'20$
$
xcss $,$.
c'10: '21$
$$
```

To prevent STAGE2 from giving an IOCH message, it has to be halted explicitly, for instance by means of the ALICE end macro. When running STAGE2 with these macro definitions and the macros:

```
chd 1,1
chd 2,0
sub 3,1,2
css 1,2
css 2,3
end "program name"
```

STAGE2 gives the same result as ML/1 above.

3 Description of the various constructs of ALICE

3.1 Values

The only primitive data types in ALEPH are integer and string. Expressions can assume only constant integral values. So one could expect that at the stage of ALICE generation, all expressions have been shrunken to values and that, for example, in ALICE a constant will be declared as follows:

```
con <representation of the constant>, <value of the constant>
```

However, at a closer inspection of constant declarations in ALEPH one comes upon machine dependent constructions such as:

```
'constant' two power 32 = 256 * 256 * 256 * 256.

'constant' cap = /A/ - /a/.

'constant' middle = (>>table - <<table) / 2.

'table' messages = ("MEDIUM" : mclw).
```

All four constants declared above (two power 32, cap, middle, mclw), illustrate a different kind of problem for a machine independent ALEPH compiler. These problems are:

1 The range of integer values;

2 The character code;

3 The division of the address space into virtual address spaces; [ALEPH MAN 4.1.4]

4 The number of machine words occupied by a string of a certain length.

ALICE provides the possibility for all expressions to be evaluated at macro time, for the obvious reason that most programs are more often executed than compiled. ALICE would have been simpler if the expressions were evaluated at run time. But then the ALEPH compiler would be inefficient and would generate inefficient code.

3.1.1 The range of integer values.

The host and target environments each have ranges of integer values that they can handle. Such a range depends on the word size of the machine and on the number of words the installer decides to allocate for an integer. These ranges can influence the evaluation of expressions.

One scheme for expression evaluation is:

Do as much calculation as possible on the host machine. When the host machine can no longer handle the value of the expression because of machine dependencies, send the intermediate results to the target machine. On the target machine the final evaluation has to be done.

This is unnecessarily complicated.

Because the host machine cannot do the complete evaluation of the expressions, no calculation at all will be done at the host machine. The range of integer values of the host machine no longer plays a role. All integral denotations are handled on the host machine as strings of digits that have to be sent to the target machine.

The ALICE processor on the target machine must be able to evaluate the expressions. To this end a very simple expression language has been added to ALICE. By means of this expression language the four problems mentioned above, can be solved.

Syntax:

```
value                  : value definition;
                         calculation.

value definition       : int denotation;
                         manifest constant;
                         char denotation;
                         string length;
                         external table length.

int denotation         : int symbol, sp,
                         location, co, integer, el.

calculation            : operator, sp, location, co,
                         valref, co, valref, el.

operator               : add symbol;
                         subtract symbol;
                         multiply symbol;
                         divide symbol.

location               : integer.
valref                 : integer.
```

Semantics:

value table:

The ALICE processor has a table in which it can store the values represented by value-definition macros and calculation macros. The values in this value table will not be redefined. The location parameter of a value macro indicates where to put a value in this table. The location parameters form an ascending row of integers. They are the indices in the value table. A result needed for further calculation is referred to by means of a valref parameter in one of the operand fields of a calculation macro. The value table is only needed at macro time.

int denotation:

The sequence of decimal digits is converted to an integer value. This value is stored in the value table at the location referred to by the location parameter.

calculation:

The result of the operation on the values referred to by the two valrefs is stored in the value table at the location referred to by the location parameter.

The operators add, subtract and multiply do what one would expect. The divide operator conforms to the ALEPH division. The result of an integer division $n = p / q$ $(q \neq 0)$ is a value n such that $p \geq (n * q)$ and $(p - n * q)$ is as small as possible.
So, $7 / 3 = 2$, $7 / (-3) = -2$, $(-7) / 3 = -3$, $(-7) / (-3) = 3$.

Example: ALICE for calculation of the value of two power 32:

```
int 1,256
mul 2,1,1
mul 3,2,2
```

The degree of optimizing done by the ALICE generator has not quite been decided yet. Somebody, in this case the author, did a beautiful job of optimizing here and it is not sure whether the compiler will go this far.

After these macros have been processed, the ALICE processor has either stored the value 4294867296 in its table at location 3, or it has emitted an error message. If the value is stored in the table, it can be used throughout the further code generation, for example as a literal in a constant source.

It is not required that the ALICE processor evaluate these expression macros at compile time. It could also let the assembler do the job or even generate code to evaluate the expressions at the start of the execution of the program. The location parameter can be used to create a name.

3.1.2 The character code

One way to get rid of the problem of the character code is to demand a specific internal character code, such as ASCII, inside every ALEPH program and consequently inside the ALEPH compiler. The standard externals 'getchar' and 'putchar' can perform a conversion between the installation character code and the internal character code. This gives the compiler the opportunity to calculate with character codes machine independently.

This at first sight attractive idea has two drawbacks. It becomes hard to read characters efficiently when they have to be handled one by one. This argument becomes important if one wants to read lines efficiently. The problem has not even completely vanished, because there will always be installations with non-ASCII characters available.

The character code will therefore be chosen by the installer and character denotations are handed over to the target machine via ALICE by means of character denotation macros.

Syntax:

```
value definition        : char denotation.

char denotation         : char denotation symbol, sp, location,
                          co, character, el.
```

Semantics:

The result of the conversion from character to character code is stored in the value table at the location referred to by the location parameter.

Special care has to be taken for characters, such as comma and space. They can make the recognition of parameters difficult for the ALICE processor, since they mix up with the delimiter structure of the macros. Therefore these characters will be declared in manifest constant macros. Manifest constant macros are used as an escape out of these exceptions, and to declare machine dependent constants.

Syntax:

```
value definition        : manifest constant.

manifest constant       : manifest constant symbol, sp,
                          location, co, manco, el.

manco                   : new line symbol;
                          same line symbol;
                          rest line symbol;
                          new page symbol;
                          max char symbol;
                          word size symbol;
                          max int symbol;
```

```
min int symbol;
int size symbol;
comma-tag symbol;
space-tag symbol;
min addr symbol;
max addr symbol;
virtual length symbol;
nil symbol.
```

Semantics:

The value denoted by the manco parameter is stored in the value table at the location referred to by the location parameter.

Example: ALICE code to calculate the value of cap:

```
chd 4,A
chd 5,a
sub 6,4,5
```

3.1.3 The address space

The data types in an ALEPH program are integer, string, and lists of these. The storage unit containing an integer is called a "virtual machine word". A virtual machine word is mapped on an (implementation dependent) number of machine words.

ALEPH programs assume the existence of a "virtual address space" for each list declared. Items in a list are identified by unique addresses, which are represented by integral values. These integral values are called "virtual addresses" and are the elements of the virtual address space associated with the list. A virtual address identifies one virtual machine word. The virtual address spaces of all lists form the "address space".

The ALEPH programmer can influence the distribution of the virtual address spaces over the address space by means of size estimates. But he cannot influence the range of integers available for the address space. This can only be done by the installer of ALICE.

Because virtual addresses are represented by integers, the range of integer values must be a superset of the address space. The size of the address space appears to be the most important measure for the range of integer values in an implementation.

The association of virtual addresses with lists is done by the compiler and it must be done in such a way that every list gets enough virtual addresses associated with it, that is, at least as many virtual addresses as it will ever have items. It is of course impossible to predict the needs of a list because the amount of virtual addresses needed for a list depends on the dynamic behaviour of the program. To help the compiler in making a reasonable guess, the ALEPH programmer writes "size estimates" in the declaration of his lists.

The virtual address space of a list consists of a contiguous set of integers. The address space consists of a contiguous set of integers, too. Because the virtual address spaces of the lists form a partition of the address space, a virtual address uniquely identifies a list and it may identify an item of that list. The question whether a virtual address points into a certain list can easily be answered by checking whether the virtual address lies between the min limit and the max limit of the list.

For stacks with an absolute size estimate and tables it is known at compile time how many virtual addresses they need. Stacks with a relative size estimate get a virtual address space proportional to their size estimate. The bounds of the address space can be declared in ALICE by means of manifest constants (min addr, max addr).

The ALEPH external table does not fit into this scheme, because the size of an external table can only be derived by inspection of the machine-dependent string that defines its contents. This must be done by the ALICE processor when an external table length macro is encountered.

Syntax:

```
external table length    : ext table length symbol, sp, location,
                           co, string, el.
```

Semantics:

The string is an exact copy of the string written in the ALEPH external table declaration. The number of virtual machine words occupied by the external table is derived by inspection of the string. The value thus obtained is stored in the value table at the location referred to by the location parameter.

It is clear that programs containing external table declarations are far from portable. The ALEPH compiler itself will therefore not contain external table declarations.

3.1.4 Strings

The way strings are handled varies from installation to installation, so it is clumsy to make strict assumptions about the way characters are packed into machine words. To define a manifest constant giving the number of characters fitting in a machine word is to assume too much about the way strings are allocated, because the number of words occupied by a string is not always the quotient of the string length and that constant.

It is, on the other hand, impossible to make no assumptions at all about the allocation of strings, for one thing because the allocation of a string influences the range of virtual addresses needed for the list the string belongs to. It is assumed that all strings of a certain length occupy the same number of words, no matter which characters they contain or how strings are allocated. This assumption can bother an installer but it has to be made, because of the dependence of address calculation and string length mentioned earlier. The value of mclw (in the example in section 3.1) can be calculated as the sum of the min limit of the table messages and the number of words occupied by the string "MEDIUM".

Syntax:

value definition	: string length.
string length	: string length symbol, sp, location, co, integer, el.

Semantics:

The number of virtual machine words occupied by a string containing a number of characters specified by the second parameter is derived. This value is stored in the value table at the location referred to by the location parameter.

3.1.5 Example

Suppose the following list declarations occur in an ALEPH program:

```
'table' powers = ( "one" :one,
                   "ten" :ten,
                   "hundred" :hundred,
                   "thousand" :thousand
                 ).

'stack' [= 5 =] digits,
        [ 30 ]  anonymous operands,
        [ 50 ]  rationals.
```

The information characterizing a list consists of:

its min limit,	(:<)
its left pointer,	(<<)
its calibre,	(<>)
its right pointer,	(>>)
its max limit.	(>:)

The following value macros may be generated to divide the address space and to gather other necessary information.

```
xxx this is a comment macro to be ignored by the ALICE processor

mcn 1,mna
xxx :<powers = mna

int 3,1
xxx <>powers = 1

add 2,1,3
xxx <<powers = :<powers + <>powers

stl 25,3
xxx string length("one")

add 21,1,25
xxx one = :<powers + string length("one")

stl 26,3
xxx string length("ten")

add 22,21,26
xxx ten = one + string length("ten")

stl 27,7
xxx string length("hundred")

add 23,22,27
xxx hundred = ten + string length("hundred")
```

```
stl 28,8
xxx string length("thousand")

add 24,23,28
xxx thousand = hundred + string length("thousand")

xxx >>powers = thousand
xxx >:powers = thousand
xxx :<digits = thousand
xxx >>digits = thousand

int 8,1
xxx <>digits = 1

add 7,24,8
xxx <<digits = :<digits + <>digits

int 29,5
xxx length(digits)

add 10,24,29
xxx >:digits = :<digits + length(digits)

xxx :<anonymous operands = >:digits
xxx >>anonymous operands = >:digits

int 13,1
xxx <>anonymous operands = 1

add 12,10,13
xxx <<anonymous operands
xxx     = :<anonymous operands + <>anonymous operands

mcn 35,mxa
xxx max addr

sub 36,35,10
xxx virtual left over = max addr - >:digits

int 37,0
xxx initialize sum relative size estimates

int 30,30
xxx relative size estimate anonymous operands

add 38,37,30
xxx sum = sum + relative size anonymous operands

int 32,50
xxx relative size estimate rationals

add 39,38,32
xxx sum = sum + relative size rationals
xxx last calculation for sum

dvd 31,36,39
```

```
xxx unit virtual addresses =
xxx              virtual left over / sum size estimates

mul 41,31,30
xxx virtual size anonymous operands
xxx     = size estimate * unit virtual addresses

add 15,10,41
xxx >:anonymous operands
xxx     = :<anonymous operands + virtual size anonymous operands

xxx :<rationals = >:anonymous operands
xxx >>rationals = >:anonymous operands


int 18,1
xxx <>rationals

add 17,15,18
xxx <<rationals = :<rationals + <>rationals

xxx >>rationals = mxa
xxx end of ALICE macros
```

RESULT:

	powers	digits	anonymous operands	rationals
:<	1	24	10	15
<<	2	7	12	17
<>	3	8	13	18
>>	24	24	10	15
>:	24	10	15	35

one	21
ten	22
hundred	23
thousand	24

The entries in these tables indicate the location of the various values.

3.2 Data

After the values have been calculated, the data initialized with these values can be declared. ALICE contains the following data declarations:

3.2.1 Integer
3.2.1.1 Constant source
3.2.1.2 Variable

3.2.2 List
3.2.2.1 List area
3.2.2.1.1 List fillings
3.2.2.2 List administration
3.2.2.3 External table

3.2.3 File
3.2.3.1 File administration

For all sorts of objects in an ALICE program, such as data objects, a unique translation-time representation is needed for reference purposes. This representation ("repr" in the grammar) is an unsigned integer. Representations can serve the ALICE processor in various ways. It is easy to make unique names (labels) from them and they serve at macro time for communication between macros. One set of integers is used to give representations to both data objects and objects in the rules part.

3.2.1 Integer

3.2.1.1 Constant source

The values calculated in the values part of the ALICE program are used for various purposes. Some are intermediate results and are no longer of interest in the data and rules part. Some are used in list and file administrations and in variable declarations. Others are used to declare constant sources.

Syntax:

```
constant source         : constant source symbol, sp, repr,
                          co, valref, el.

repr                    : integer.

valref                  : integer.
```

Semantics:

A constant source declares a constant actual parameter of a call, before it is going to be used as such in the rules part of ALICE. The value of the constant source is in the value table at the location referred to by the valref parameter.

Installation hints:

Depending on the implementation and the facilities the assembler has to offer, the ALICE processor may act in various ways:

hint 1.

If the implementation views constant sources just as variables that happen not to change their value, the ALICE processor will generate assembly data declarations from the constant sources. The representation will then serve to generate a unique label for the constant. When code for picking up the constant actual parameter has to be generated, this can be done easily because the ALICE macro for doing this contains the representation of the constant source again.

hint 2.

If the assembler offers a suitable literal mechanism, no code need be generated from the constant source macros. When the code to pick up the constant actual parameter is generated, the valref also supplied in the macro for doing this will serve to refer to the value in the value table. From this value a literal can be generated.

hint 3.

In most cases the assembler provides a literal mechanism for numbers up to a certain maximum value, for instance because both opcode and literal have to fit in one machine word. A mixture of 1. and 2. will then do the job.

Example:

ALEPH: 0 -> counter

ALICE	macro time action
values part	
int 5,0	hint 1. and 2. store value 0 in symbol table at location 5
data part	
css 21,5	hint 1. generate label from representation get value from symbol table location 5 generate data declaration hint 2. do nothing
rules part	
lvc 21,5	hint 1. use repr to refer to data declaration hint 2. get value from symbol table at location 5 use value to generate a literal

3.2.1.2 Variable

From a variable macro the ALICE processor will generate an assembler data declaration. The value used to initializing the variable has been calculated in the values part.

Syntax:

```
variable declaration     : variable symbol, sp, repr, co,
                           valref, co, repr, string, el.
```

Semantics:

A variable declaration causes the allocation of one virtual machine word represented by the first parameter and initialized with the value in the value table at the location referred to by the second parameter. The third parameter is the representation of the next variable to be declared. The fourth parameter contains the ALEPH variable-tag in string quotes. If the third parameter is equal to zero, there is no next variable declaration.

Installation hints:

The first parameter will be used to generate a label, the second to generate a data declaration. If a global dump is requested (see section 3.4 status macro), the third parameter serves to generate a pointer to the next variable and the fourth parameter serves to generate a string containing the ALEPH variable-tag.

Example:

ALEPH: ′variable′ counter = 0.

```
ALICE                   macro time action

values part

int 5,0                 store value 0 in value table
                        at location 5
```

data part

```
var 39,5,40,"counter"   generate label from representation 39
                        get value from symbol table location 5
                        and generate data declaration
                        if global dump is requested generate
                        pointer to next variable and
                        string with the name of this variable
```

3.2.2 List

3.2.2.1 List area

The list areas are used to set up the physical address space in which the lists are floating, and to initialize these lists with their initial list filling. For each ALEPH list a list area will be declared giving the initial fillings consisting of integer values already calculated in the values part, or of strings. Information about the list type and the size of the virtual address space is supplied at the beginning and end of the area. A list is called "breathing" if its size can change at run time, that is if it is a stack with a relative size estimate. When and how the physical address space is actually set up depends on the implementation.

Syntax:

```
list areas            : list area,
                        [list areas].


list area             : list symbol, sp, repr, co,
                        list type, co,
                        valref, el,
                        [list fillings],
                        end list symbol, sp, repr, co,
                        list type, co, valref, el.

list fillings         : list filling, [list fillings].

list filling          : int fill symbol, sp, valref, el;

                        string fill symbol, sp, string, el;

                        fallow symbol, sp, valref, el.
```

Semantics:

The parameters of the first macro of a list area, a list macro, have the following meaning:

First parameter, repr: the representation of the list.

Second parameter, list type: an integer

0:	not breathing, no background
1:	not breathing, background
2:	breathing, no background
3:	breathing, background

If background is specified there is a background pragmat identifying the list in the ALEPH program.

An int fill macro causes the allocation of a virtual machine word, initialized with the value in the value table at the location referred to by the valref parameter.

A string fill macro causes the allocation of a number of virtual machine words, initialized with a string value denoted by the string parameter.

A fallow macro causes the allocation of a number of uninitialized virtual machine words. This number is found in the value table at the location referred to by the valref parameter.

The three parameters of the last macro of a list area, an end list macro, have the same meaning as those of a list macro.

Installation hints:

Depending on the target machine, the operating system, and the rest of the ALICE implementation, the physical address space can be implemented in various ways:

hint 1.
Allocate a contiguous chunk of storage for the lists. Put all initial list fillings of the lists right behind each other in the chunk of storage. Let a run-time routine shuffle the lists.

hint 2.
Allocate one chunk of storage as in hint 1. A non breathing list gets a piece of that chunk just fitting its needs, while a breathing list gets a number of extra uninitialized storage locations. This can be either a constant number or a number proportional to the number of virtual addresses associated with the list. The list shuffling is done by the same run-time routine as in hint 1.

hint 3.

Implement a paging scheme for the lists. Every list gets one (or more) page(s) in core, while a complete virtual address space is allocated on disk.

hint 4.

If the operating system provides a suitable virtual addressing mechanism, map the virtual ALEPH addresses on the virtual machine addresses. In this case the physical address space is equal to the virtual address space.

hint 5.

How to deal with the background information depends heavily on the implementation. It is formally correct to ignore it completely. If, for instance, the physical address space is implemented according to hint 3, the background information has no meaning. Only in cases where lists can both be allocated in core and on background, the background information could be of use.

hint 6.

Most implementers will have to deal with machines having a von Neumann store, whether they like it or not. In that case all lists will probably be allocated in one contiguous physical address space. A list-shuffling routine is then needed as part of the run-time system to prevent the breathing lists from bumping into each other. Because this problem is very likely to arise for a lot of machines, such a list-shuffling routine is available in ALEPH (and consequently in ALICE).

3.2.2.2 List administration

A list administration macro is used to generate the administration information of a list. The correspondence between a virtual address and a physical address must be derived from this information, no matter how the lists are allocated. The limits must be available in the administration, because they can be used as actual parameters at run time. The type, the limits and the maximal virtual address are inspected when a list is extended. The limits are updated when the size of the list is changed.

Syntax:

```
list administration      : list adm symbol, sp,
                           list info, el.

list info                : repr, co, list type, co,
                           valref, co,
                           valref, co,
                           valref, co,
                           valref, co,
                           valref, co,
                           repr, co,
                           string.
```

Semantics:

The first parameter of a list administration is the representation of the list.

A list administration macro causes the allocation of a data structure called "list administration" such that at least the following information can be kept in it:

The list type, specified by the list type parameter. This parameter has the same meaning as the list type parameter of a list macro.

The lowest virtual address associated with the list: called "virtual min limit". This value is found in the value table at the position referred to by the third parameter.

The highest virtual address associated with the list: called "virtual max limit". This value is found in the value table at the position referred to by the fourth parameter.

The virtual address of the left most block in the list: called "virtual left". This value is found in the value table at the position referred to by the fifth parameter.

The virtual address of the right most block in the list: called "virtual right". This value is found in the value table at the position referred to by the sixth parameter.

The calibre of the list, that is: the number of elements in one block of the list. This value is found in the value table at the position referred to by the seventh parameter.

Information to convert a virtual address associated with the list to the corresponding physical address.

The address of the next list administration. The eighth parameter contains the representation of the next list administration or zero.

The ALEPH list-tag. The ninth parameter contains the ALEPH list-tag in string quotes.

Installation hints:

hint 1.
The representation of the list can be used to generate a label of the list administration, and to make the address of the list area (same representation) available in the list administration.

hint 2.
The ALEPH list-tag is needed for the implementation of data files (see installation hints of file administration).

hint 3.
In most implementations not the whole virtual address space is mapped in core. In those cases in comes handy to have two extra limits, for instance called "bumpmin" and "bumpmax", in the list administration, indicating which portion of the virtual space of the list is mapped in core. If, for instance, an implementation according to hint 3. in section 3.2.2.1 is chosen, bumpmin and bumpmax are the virtual addresses of the beginning and end of the core page.

Example:

Suppose the lists are allocated in one chunk of contiguous storage.
A list is labeled: l<repr of list>
That means: the virtual machine word following the virtual machine word labeled l<repr> is the left most element of the left most block of the (initial) list. The virtual address of the left most block of a list is the virtual left of the list. To convert a virtual address to a physical address, the mixed expression:

```
        l<repr> - virtual left + calibre
```

is added to that virtual address. This mixed expression is also kept in the list administration.

3.2.2.3 External table

An external table is declared by one macro, very much alike a list administration macro. The external table macro has one more parameter: the ALEPH string containing information about the filling of the table. What, exactly, is in the string depends on the particular implementation and cannot be described machine independently. It could be a number of data declarations in assembly language, or an external reference to be satisfied by the local loader, or an access routine. The ALICE processor must generate an administration, whose layout does not differ from the administration of a normal table, because external tables and tables can both be passed as actual parameters to the same formal table parameter.

Syntax:

```
external table decl      : external table decl symbol, sp,
                           list info, co,
                           string, el.
```

Semantics:

An external table macro causes the allocation of a list administration and a list area. The first nine parameters (list info), have the same meaning as the parameters of a list administration macro. The last parameter is an exact copy of the ALEPH string of the external table. From this parameter a list area must be generated.

Installation hints:

A portable ALICE program, such as the ALEPH compiler, will not contain external table declarations. The implementation of this macro is therefore not needed.

3.2.3 File

3.2.3.1 File administration

A file administration macro serves to generate book keeping information and buffers for a file. The file administrations are chained, just like the list administrations, so that they can be easily initialized or post processed.

Syntax:

```
file administrations     : file administration,
                           [file administrations].

file administration      : begin file adm symbol, sp, file info, el,
                           [pointer area],
                           [numerical area],
                           end file adm symbol, sp, file info, el.

file info                : repr, co,
                           file type, co,
                           repr, co,
                           string.

file type                : integer.


numerical area           : numerical symbol, sp,
                           valref, co,
                           valref, el,
                           [numerical area].

pointer area             : pointer symbol, sp, repr, el,
                           [pointer area].
```

Semantics:

The parameters of the first macro of a file administration, a begin file administration macro, have the following meaning:

First parameter, repr: the representation of the file.

Second parameter, file type: an integer

0:	scratch charfile
1:	scratch datafile
2:	input charfile
3:	input datafile
4:	output charfile
5:	output datafile
6:	input/output charfile
7:	input/output datafile

If input is specified, the only permitted action is reading.
If output is specified, the only permitted action is writing.
If input/output is specified, both reading and writing are permitted.
If scratch is specified, the file is an input/output file which will not be available after the execution of the program.

If charfile is specified, the items to be read or written are characters, represented by (small) integers according to the character code chosen by the installer.

If datafile is specified, the items to be read or written consist of an integer value and an indication about its meaning. This indication is either "numerical" in which case the integer value stands for itself, or it is the ALEPH name of a list in which case the integer value is an offset from the left of the list.

Third parameter, repr: the representation of the next file administration or zero.

Fourth parameter, string: the ALEPH string from the file declaration.

Pointer macros and numerical macros specify a number of "zone"s. A zone is a contiguous set of integers.
A numerical macro specifies a zone by supplying two valrefs. The first valref refers to the minimum of the zone, while the second refers to the maximum of the zone.
A pointer macro specifies a zone by supplying the representation of a list. The zone is the virtual address space of the list.

When a file is used for reading, each item delivered will belong in one of the zones.

When a file is used for writing, each item offered must belong in one of the zones.

If the file type specifies charfile, there will be no pointer macros.

The parameters of the last macro of a file administration, an end file administration macro, have the same meaning as those of a begin file administration macro.

Installation hints:

hint 1.
If the operating system provides a way to handle character files efficiently, for instance by packing a number of characters in one word, this facility can be used for the implementation of charfiles.

hint 2.
The data item of a datafile can be implemented in two ways.
a) The data item can consist of an integer value and a Boolean value indicating that the integer must be considered as a virtual address. In that case the file will begin with heading information consisting of triples:

file name	virtual min limit	virtual max limit.

b) The data item can consist of two integer values. The first integer value is a stripped virtual address, that is a virtual address minus the min limit of the list it pointed in. The second integer indicates in which list the virtual address pointed. Here the the file will begin with heading information consisting of file names.

When a data item is read, the following actions have to be performed:

1. Find out whether the data item is a pointer (The Boolean is <u>true</u> or the second integer is positive) or not.

2. If it is a pointer, identify the list name.
In the first case this is done by comparing the virtual address with the virtual min limits and max limits of the heading information. In the second case it is done by selecting the right file name from the heading information.

3. Find the identical list name in the list administrations.
If the list name from the file is not identical to one of the list names in the program, the situation is erroneous.

4. Relocate the pointer.
In the first case this is done by subtracting the old virtual min limit available in the file heading and adding the virtual min limit of the identified list. In the second case this is done by adding the virtual min limit of the identified list.

3.3 Rules

3.3.1 Introduction

In order to indicate the reasons behind the various ALICE constructs, the semantics of a call in ALEPH will be discussed first. The ALEPH version of a subroutine call differs from calls in most programming languages in that the result of a call influences the flow of control. An affix form (subroutine call) identifies a rule (subroutine) to be executed and describes the actual parameters. The execution of the rule starts with copying the actual input parameters of the affix form to the private stack frame of the rule. Then the actual rule is executed. The execution of the actual rule can succeed or fail. If it succeeds the output parameters are restored to the output parameters of the affix form. If it fails the output parameters will not be restored. The affix form succeeds or fails if the rule called succeeds or fails, respectively.

The basic building blocks of an ALEPH rule are called "member"s. The term member covers affix forms as well as language primitives such as assignment. Primitives obey the same laws in terms of flow of control as affix forms. Every member is connected to two positions in the rule: a position for a successful execution of the member and a position for a failing execution. Such a position in a rule can either be a member or the end of a rule body. In the latter case the execution of the rule has come to an end, and the result is passed to its caller. If the position is a member, the success or failure of the member is of no further interest; control simply continues. A member calling an action or a function needs only be connected to one position, because actions and functions never fail.

Compound members have not been taken into account yet. In the semantics of ALEPH a compound member is merely a call to an implicitly defined rule. This does not mean that compound members are translated as such. There are no compound statements in ALICE. An ALEPH compiler must use other facilities to translate them. Consider, for example, the ALEPH rule

```
'predicate' p: q1, a1, a2;
               (q2; q3), a3, q4, :p;
               q5.
```

where the qi's stand for members that can fail and the ai's for members that cannot fail. The table below states the connections between the various members of this rule.

	true address	false address
q1	a1	q2
q2	a3	q3
q3	a3	q5
q4	:p	.
q5	.	.
a1	a2	
a2	.	
a3	q4	
:p	q1	

The dot "." means the end of the rule body.

ALEPH rules are translated to ALICE rules. An ALICE rule consists of a rule head, a rule body, and a rule tail. The rule head identifies the rule and copies input parameters to the private stack frame of the rule. The rule body consists of a sequence of statements with explicit flow of control using labels, jumps, and true and false-addresses in calls. Rules and labels have unique integer representations, like all ALICE objects. The rule tail either restores output parameters from the private stack frame of the rule and returns success to the caller or it returns failure to the caller.

The following is a sketch of the translation of the ALEPH 'predicate' p above. The macros for rule identification, call, and rule termination have not been worked out in detail. They represent groups of ALICE macros. A group of ALICE macros describing a call has been written down symbolically as:

```
call <rule called>, <true address>, <false address>
```

where the false address is omitted if the call cannot fail and the representation of the address of the next sequential instruction is zero.

```
rule <p>
label 1
call <q1>,0,4
call <a1>,0
call <a2>,2
label 4
call <q2>,5,0
call <q3>,0,6
label 5
call <a3>,0
call <q4>,0,3
jump 1
label 6
call <q5>,0,3
success-tail 2
fail-tail 3
```

3.3.2 The ALICE abstract machine

The semantics of ALICE will be described in terms of an abstract machine, the ALICE ABSTRACT MACHINE (A.A.M.). The parts of the A.A.M. will be introduced now.

The A.A.M. has a memory in which machine code and the data structures described in section 3.2 are allocated. These data structures are addressed via their representation.

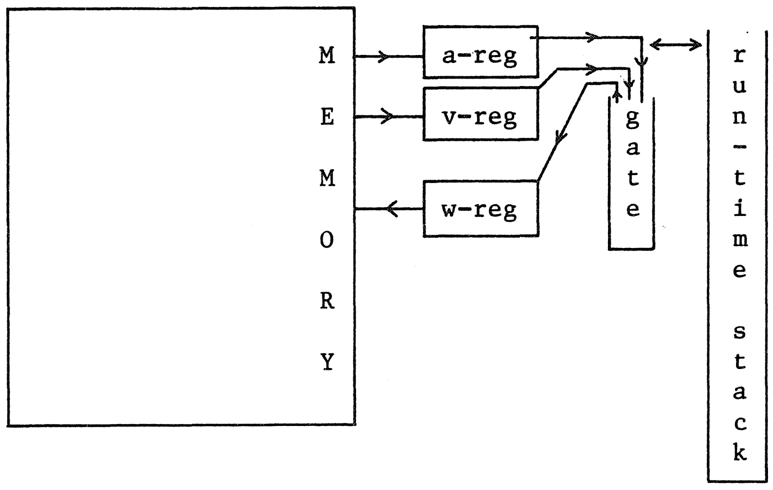
Furthermore, the A.A.M. has a stack, called the "run-time stack". On this run-time stack the formal and local parameters of the rules in execution, and the return addresses to their callers are kept. Formals and locals are addressed by means of integers, called "stack position".

A sequence of machine words is used to pass parameters to and from the run-time stack. In real machines this can be a contiguous sequence of memory locations, or a number of registers, or the run-time stack itself. In the A.A.M. a special device is used for this purpose. It is called the "gate", and it is manipulated in stack fashion. Elements on the gate have integer addresses starting from one. These addresses are, strictly spoken, superfluous because the gate is treated like a stack, but when the gate is mapped on a real machine these integers can be practical. When the gate is used to pass parameters to the run-time stack, it is called the "input gate". When it is used to pass parameters back to the caller, the gate is called the "output gate".

To load the actual parameters from memory and to stack them on contiguous positions of the gate, two abstract machine registers are used. One register, the "v-register" can hold an integer value. The other one, called "a-register", can hold the address of an administration. The description of an input parameter consists of one or more statements to load the a-register or v-register, followed by one statement to store the contents of the a-register or v-register on a specific position of the gate.

During the execution of a call, one position on the gate corresponds to one position on the run-time stack. For some implementations it may well be that the position on the run-time stack must be known by the caller, for instance because the actual parameters are pushed directly on the run-time stack. Therefore, both gate position and run-time stack position are given in the statements to store the contents of the a-register or v-register on the gate. Gate position and stack position form a pair of integers, which are always supplied together. This pair of integers is called a "formal".

To unstack an output parameter from the gate another abstract machine register is needed. This register, called the "w-register" can hold an integer value. The description of an output parameter consists of one or more statements to load the w-register, eventually with the aid of the a-register or v-register, followed by one statement to store the contents of the w-register into memory.



The ALICE abstract machine

3.3.3 Outline of an ALICE rule

Syntax:

```
rule decl                : rule head, rule body, rule tail.

rule head                : rule id,
                           stack frame,
                           [copies from input gate].


rule body                : statements.

statements               : statement,
                           [statements].

statement                : call;
                           extcall;
                           primitive.

rule tail                : success tail,
                           [fail tail].
```

The ALICE macros making up a rule are divided in three groups: rule head macros, rule body macros, and rule tail macros. The rule head macros identify the rule, supply debugging information and describe the correspondence between the input gate and the run-time stack frame of the rule. The rule body consists of a sequence of statements. A statements is either a call to another ALICE rule, or a call to an external rule, or an ALICE primitive. If the rule cannot fail, the rule tail consists of one section: a success tail. If the rule can fail, the rule tail consists of a success tail and a fail tail. In the success tail the correspondence between run-time stack frame and output gate, unstacking, and returning success is described. In the fail tail unstacking and returning failure is described.

3.3.4 Rule head

Syntax:

```
rule id                  : rule id symbol, sp, repr, co,
                           rule type, co, recursion, co, string, el.

rule type                : integer.

recursion                : integer.

stack frame              : stack frame symbol, sp,
                           integer, co, integer, el.

copies from input gate   : copy from input gate,
                           [copies from input gate].
```

```
copy from input gate     : copy from input gate symbol, sp,
                           formal, el.

formal                   : integer, co, integer.
```

Semantics:

The parameters of a rule id macro have the following meaning:
First parameter, repr: the representation of the rule.
Second parameter, rule type: an integer:
 0: the rule cannot fail
 1: the rule can fail
Third parameter, recursion: an integer:
 0: non recursive rule
 1: recursive rule
Fourth parameter, string: The ALEPH rule tag and formal affix sequence in quotes.

A stack frame macro declares the number of formals and locals to be allocated on the run-time stack. The first parameter denotes the number of formals, the second denotes the number of locals.

For every input parameter there will be one copy from input gate macro. A copy from input macro causes popping of one value from the gate and copying it to a position of the run-time stack. The first parameter denotes the position one the gate, the second parameter denotes the position one the run-time stack. The position on gate parameters form a descending row, from the number of input parameters to one.

Installation hints:

hint 1.
The representation of the rule serves to generate a label. The string of a rule id macro is needed when debugging information must be generated, because a global dump is requested (see section 3.4 status macro).

How the stack frame and copy from input gate macros are used depends on the implementation of the calling mechanism. Throughout this section two models for the implementation of the calling mechanism will be described: In the first model the role of the gate is played by registers r1, r2 , ... In the second model the parameters are passed directly on the run-time stack, that is, the way from v-register (or a-register), via gate, to run-time stack is cut short.

hint 2.
If the parameters are passed in registers, the stack frame macro causes the allocation of a number of machine words for the formals and locals on the run-time stack, and the pushing of the return address, passed by the caller, on the run-time stack.
A copy from input gate macro causes a move from the register indicated by the first parameter to the run-time stack position addressed by the second parameter.

hint 3.

If the parameters are passed directly on the run-time stack, the caller will do that. The only thing that may be done in the rule head is pushing the return address on the run-time stack.

3.3.5 Rule tail

Syntax:

```
rule tail              : success tail,
                         [fail tail].

success tail           : success tail id,
                         output gate creation,
                         [restores to output gate],
                         unstack and return true.

success tail id        : success tail id symbol, sp, repr, co,
                         rule type, co, recursion, el.

output gate creation   : output gate symbol, sp, integer, el.

restores to output gate : restore to output gate,
                         [restores to output gate].

restore to output gate : restore to output gate symbol, sp,
                         formal, el.

unstack and return true : unstack and return symbol, sp,
                         integer, co, integer, co, true symbol, el.

fail tail              : fail tail id,
                         unstack and return false.

fail tail id           : fail tail id symbol, sp, repr, co,
                         rule type, co, recursion, el.

unstack and return false: unstack and return symbol, sp,
                         integer, co, integer, co, false symbol, el.
```

Semantics:

The parameters of a success tail macro have the following meaning:
First parameter, repr: the representation of the success tail.
Second parameter, rule type: see rule id macro.
Third parameter, recursion: see rule id macro.

An output gate macro declares the number of output parameters, and consequently the size of the output gate.

For every output parameter there will be one restore to output gate macro. A restore to output gate macro causes stacking of one formal output parameter on the gate. The first parameter denotes the position on the gate, the second parameter denotes the position on the run-time stack.

The parameters of an unstack and return macro, have the following meaning:
First parameter, integer: the number of formals on the run-time stack.
Second parameter, integer: the number of locals on the run-time stack.
Third parameter, true or false: result of the execution of the rule.

The parameters of a fail tail macro have the following meaning:
First parameter, representation of the fail tail.
Second parameter, rule type: see rule id macro.
Third parameter, recursion: see rule id macro.

Installation hints:

hint 1.
The representations of success tail and fail tail serve to generate labels. In the code generated by jump macros and call macros there will be jumps to these labels.

hint 2.
If the parameters are passed in registers, the create output gate macro informs how many registers will be needed for that. A restore to output gate macro causes a move from the run time stack position addressed by the second parameter to register r<first parameter>.

hint 3.
If the parameters are passed on the run-time stack, the caller will restore the output parameters.

hint 4.
The result of the execution of a rule (success or failure) can be reported to the caller in two ways:
a) There is a register or memory location allocated for this purpose. The rule tail assigns true or false to it, and the caller tests it.
b) Depending on success or failure the rule returns to a different address. For instance, in the fail tail a jump to the return address is generated, while in the success tail a return to the next instruction is generated.

hint 5.
The tricks in hint 4 are only needed if the rule can fail (rule type).

Example:

ALEPH:

```
'predicate' p + >i + o> + >io> - 1:
```

p is a recursive rule.

The table below shows the corresponding ALICE rule head and tail. Instead of the three-letter representations for the macro names, more readable tags are used.

```
rule-id 1000,1,1,"p+>i+o>+>io>"
stack-frame 3,1
copy-from-gate 2,3
copy-from-gate 1,1

succ-tail-id 1015,1,1
output-gate 2
restore-to-gate 1,3
restore-to-gate 2,2
unstack-return 3,1,true
fail-tail-id 1016
unstack-return 3,1,false
```

3.3.6. Calling a rule

Calling a rule proceeds as follows:

```
1. Identification of the rule to be called.
2. Stacking the actual input parameters on the (input) gate.
3. (subroutine) jumping to the rule.
4. Upon return:
   if the rule has failed
      continuing at the false address
   else
      unstacking the output parameters from the output gate
      continuing at the true address
```

The macros concerning parameter passing will be discussed in detail in the next section.

Syntax:

```
call                   : call id,
                         input gate creation,
                         target stack frame,
                         [copies to input gate],
                         scall or fcall,
                         [restores from output gate],
                         link.


call id                : call id symbol, sp, repr, co, rule type, co,
                         recursion, el.

input gate creation    : input gate symbol, sp, integer, el.

target stack frame     : target stack frame symbol, sp,
                         integer, co, integer, el.

scall or fcall         : scall symbol, sp, repr, el;
                         fcall symbol, sp, repr, co, repr, el.

link                   : link symbol, sp, repr, el.
```

Semantics:

The parameters of a call id macro have the following meaning:

First parameter, repr: representation of the rule to be called. Second parameter, rule type: an integer:

0: the rule to be called cannot fail
1: the rule to be called can fail

Third parameter, recursion: an integer:

0: the rule to be called is not recursive
1: the rule to be called is recursive

An input gate macro declares the number of input parameters, and consequently the size of the input gate.

A target stack frame macro declares the number of formals and locals of the rule to be called. The first parameter denotes the number of formals, the second denotes the number of locals.

If there is a scall macro, the rule to be called cannot fail (rule type = 0). If there is a fcall macro the rule to be called can fail (rule type = 1). The first parameter of a scall macro or fcall macro is the representation of the rule to be called. The second parameter of a fcall macro is the representation of the false address of the call.

The parameter of a link macro is the true address of the call.

Installation hints:

hint 1.
If the parameters are passed in registers, the input gate macro informs how many registers will be needed for that.

hint 2.
If the parameters are passed directly on the run-time stack, the target stack frame macro supplies the necessary information to build up the stack frame of the rule to be called.

hint 3.
The scall macro will generate code to pass the return address to the rule to be called, and to jump to the rule.

hint 4.
The fcall macro will generate code to pass the return address to the rule to be called, and to jump to the rule. The code on the return address depends on how the result (success or failure) of the execution of the called rule is returned to the caller. If it is returned in a register or memory location (hint 4 a in the previous section), there will be a test and a conditional jump to the false address. In the other case (hint 4 b in the previous section) there will be just a jump to the false address.

3.3.7 Parameter passing

3.3.7.1 Actual input parameters

To stack the actual input parameters on the gate, the v-register and a-register are used.

Syntax:

```
copies to input gate     : copy to input gate,
                           [copies to input gate].

copy to input gate       : copy val to input gate;
                           copy addr to input gate.

copy val to input gate   : load val in v reg,
                           copy v reg to input gate.

load val in v reg        : load simple in v reg;
                           load indexed element in v reg.

load simple in v reg     : load constant in v reg;
                           load variable in v reg;
                           load stack var in v reg;
                           load limit in v reg.


copy v reg to input gate: copy v reg symbol, sp, formal, el.

copy addr to input gate : load addr in a reg,
                           copy a reg to input gate.

load addr in a reg       : load global addr in a reg;
                           load stack var in a reg.

copy a reg to input gate: copy a reg symbol, sp, formal, el.

formal                   : integer, co, integer.



load constant in v reg   : loadv constant symbol, sp, repr, co, valref, el.

load variable in v reg   : loadv variable symbol, sp, repr, el.

load stack var in v reg : loadv stack var symbol, sp,
                           position on stack, el.
position on stack        : integer.

load limit in v reg      : load addr in a reg,
                           loadv limit symbol, sp, limit type, el.
limit type               : integer.
```

```
load indexed element
        in v reg       : indexed input parameter symbol, el,
                         load simple in v reg,
                         load list element in v reg sequence.

load list element in
        v reg sequence : load list element in v reg,
                         [load list element in v reg sequence].


load list element
        in v reg       : load addr in a reg,
                         loadv list elem symbol, sp, integer, el.


load global addr
        in a reg       : loada global symbol, sp, repr, el.

load stack var
        in a reg       : loada stack var symbol, sp,
                         position on stack, el.
```

Semantics:

A loadv constant macro causes the value of the constant source, described by the parameters of the macro, to be loaded in the v-register. The valref in the loadv constant macro refers to a location in the value table. The repr in the loadv constant macro is the representation of the constant source.

A loadv variable macro causes the value of the variable, represented by the repr parameter of the macro, to be loaded in the v-register.

A loadv stack var macro causes the value of the stack variable (formal or local parameter) of the caller, addressed by the position on stack parameter of the macro, to be loaded in the v-register.

A loada global macro causes the address of the administration of a list or file, represented by the repr parameter of the macro, to be loaded in the a-register.

A loada stack var macro causes the value of the stack variable (formal or local parameter) of the caller, addressed by the position on stack parameter of the macro, to be loaded in the a-register.

A loadv limit macro causes the value of a limit to be loaded in the v-register. The a-register contains the address of the list administration from which this value must be retrieved. The limit type parameter indicates which limit has to be retrieved:
0: left
1: right
2: calibre

An indexed input parameter macro has no effect on the ALICE abstract machine. It announces that an list element is going to be loaded in the v-register.

A loadv list elem macro causes the v-register to be loaded with a list element. The a-register contains the address of the list administration. The v-register contains the virtual address of the block containing the list element. The integer parameter indicates which element of the block has to be selected:

0: right most element
i: (i-1)-th right most element

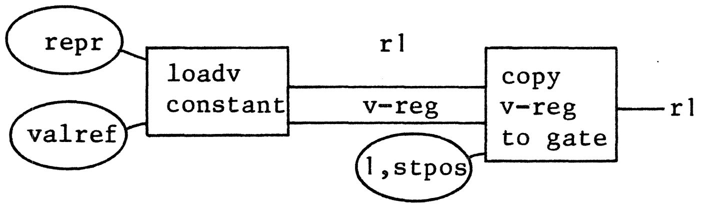
A copy v reg to input gate macro causes the contents of the v-register to be stacked on the gate. The first integer parameter of this macro denotes on which gate position the v-register must be stacked. The second integer parameter denotes on which run-time stack position the contents of the v-register will be copied.

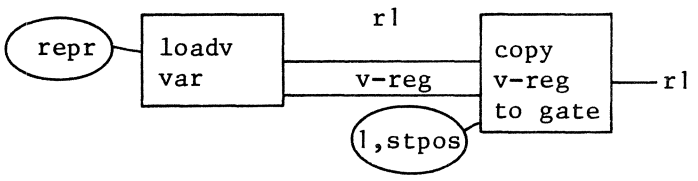
A copy a reg to input gate macro causes the contents of the a-register to be stacked on the gate. The first integer parameter of this macro denotes on which gate position the a-register must be stacked. The second integer parameter denotes on which run-time stack position the contents of the a-register will be copied.

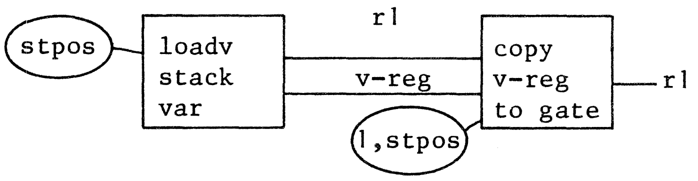
The value of the a-register and v-register can not be used more than once.

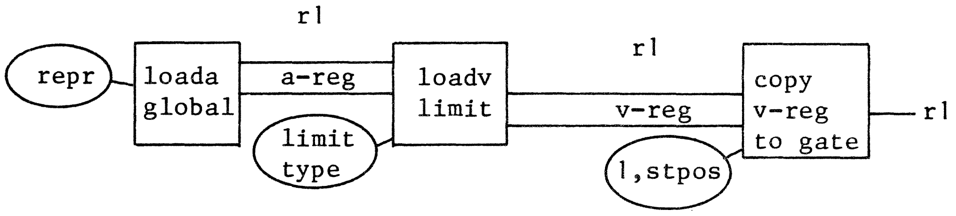
Installation hints:

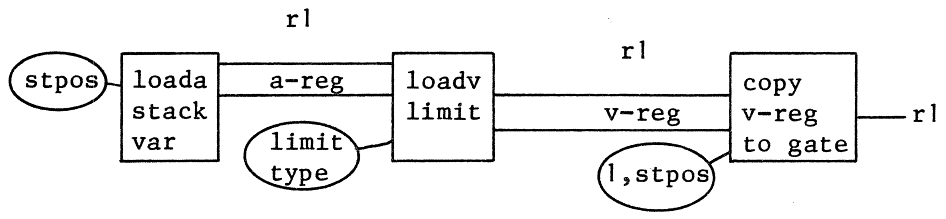
hint 1.
If the parameters are passed in registers use can be made of the stack wise treatment of the gate. The register allocated for the v-register or a-register can be the same as the one that will be allocated for the next position of the gate, so that a copy to gate macro will generate no code at all. In that case it is necessary to know which register will be copied to the gate before the copy to gate macro is read. Because of the simple structure of ALICE this can indeed be known. The figures below shows which registers have to be allocated for the a-register and v-register when the first parameter is stacked on the gate.

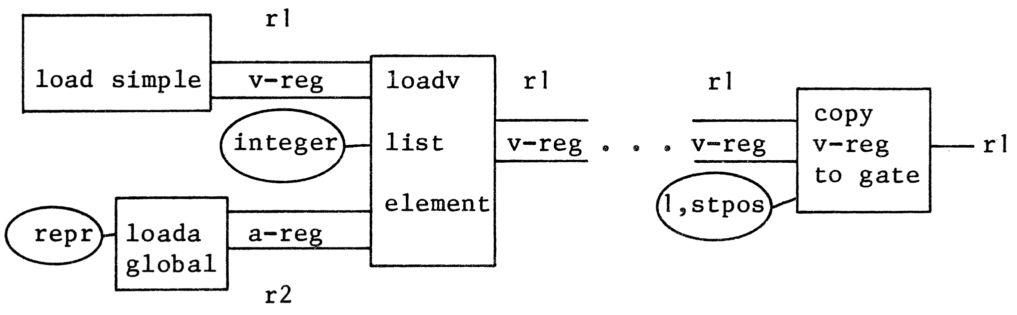

repr
valref
loadv
constant
r1
v-reg
1,stpos
copy
v-reg
to gate
r1

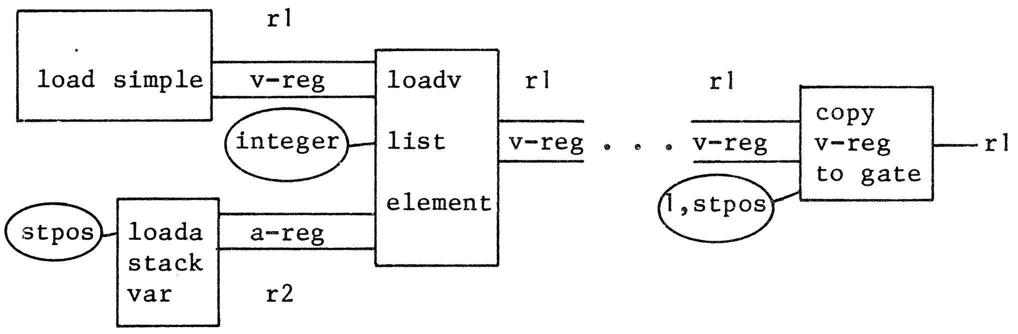

repr
loadv
var
r1
v-reg
1,stpos
copy
v-reg
to gate
r1


stpos
loadv
stack
var
r1
v-reg
1,stpos
copy
v-reg
to gate
r1


repr
loada
global
r1
a-reg
limit
type
loadv
limit
r1
v-reg
1,stpos
copy
v-reg
to gate
r1


r1
stpos
loada stack var
a-reg
limit type
loadv limit
r1
v-reg
1,stpos
copy v-reg to gate
r1


r1
load simple
v-reg
integer
repr
loada global
a-reg
r2
loadv list element
r1
v-reg . . .
r1
v-reg
1,stpos
copy v-reg to gate
r1


r1
load simple
v-reg
integer
stpos
loada stack var
a-reg
r2
loadv list element
r1
v-reg . . .
r1
v-reg
1,stpos
copy v-reg to gate
r1

hint 2.

If the parameters are passed directly on the run-time stack and the machine provides moves from memory to memory, no registers at all have to be allocated for the a-register and v-register. for instance, from the macros:

```
loadv-var       11
copy-v-reg      1,3
```

an ALICE translator could generate:

```
move v11,stackpointer-3
```

hint 3.

Because the values of the a-register and v-register do not have to be remembered, efficient code can be generated. For instance when a limit has to be retrieved, the register playing the role of the a-register firstly, can be used to play the role of the v-register later on (see hint 1).

3.3.7.2 Actual output parameters

To store an output gate element in an indexed element first the address of that indexed element must be calculated. This is described by the same macros as for an indexed input parameter. From the above it is clear that both a-register and v-register are needed for this purpose. This is why a special output register, the w-register, is needed in the abstract machine.

Syntax:

```
restores from
        output gate     : restore from output gate,
                          [restores from output gate].

restore from output gate: copy gate elem to w reg,
                          store w reg sequence,
                          free w reg.

store w reg sequence    : store w reg, [store w reg sequence].

copy gate elem to w reg : loadw symbol, sp, formal, el.

store w reg             : store w reg in variable;
                          store w reg in list element;
                          store w reg in stack var.

store w reg in variable : storew variable symbol, sp, repr, el.
store w reg in
        list element      : indexed output parameter symbol, el,
                            load simple in v reg,
                            [load list element in v reg sequence],
                            load addr in a reg,
                            storew list element symbol, sp, integer, el.

store w reg in
        stack var         : storew stack var symbol, sp,
                            position on stack, el.

free w reg                : free w reg symbol, el.
```

Semantics:

A copy gate elem to w reg macro causes the top element of the gate to be popped from the gate and to be loaded in the w-register. The first integer parameter denotes the position of the output parameter on the gate, the second integer parameter denotes the position on the run-time stack.

A storew variable macro causes the w-register to be stored in the variable, represented by the repr parameter of the macro.

A storew stack var macro causes the w-register to be stored in a stack variable (formal or local parameter of the caller), addressed by the position on stack parameter of the macro.

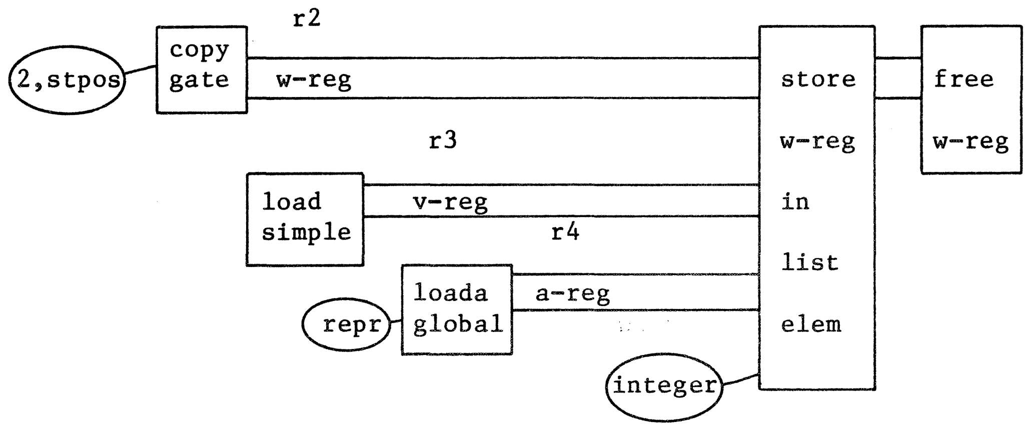
An indexed output parameter has no effect on the ALICE abstract machine.

A storew list element macro causes the w-register to be stored in a list element. The a-register contains the address of the administration of the list. The v-register contains the virtual address of the block containing the list element. The integer parameter of the macro indicates which element of the block has to be selected just as the integer parameter of a loadv list elem macro.

Installation hint:

hint 1.

If the parameters are passed in registers, the register allocated for the w-register can the same as the register allocated for the top of the gate. If there are, e.g., two output parameters, the gate consists of r1 and r2. When restoring the first output parameter, which is located in r2, the role of the w-register is played by r2, too. The figure below shows the register allocation in case of a list element as actual output parameter.



Example:

```
ALEPH: 'predicate' p + >i + o> + >io> - 1:

'stack' [1](s1,s2)st.

'variable' res = 0.
```

The table below shows the ALICE call macros corresponding to the ALEPH affix form "p + <<st + res + s1*st[>>st]". Instead of the three-letter ALICE symbols, more readable tags are used.

```
call-id         1000,1,1
input-gate      2
target-st-frame 3,1
loada-global    50
loadv-limit     0
copy-v-reg      1,1
indexed-input-parameter
loada-global    50
loadv-limit     1
loada-global    50
loadv-list-el   0
copy-v-reg      2,3
fcall           1000,510
copy-gt-to-w    2,2
storew-var      100
freew
indexed-output-parameter
copy-gt-to-w    1,3
loada-global    50
loadv-limit     1
loada-global    50
storew-list-el  0
freew
link            0
```

3.3.8 Primitive rules

ALICE has "externals" and "primitives". They can be considered as the operations of ALICE. Primitives are not declared, externals are. The ALICE-primitives are the operations that have parameters that cannot be described as the parameters of ALICE calls. The externals have exactly the same parameter descriptors as the ALICE calls.

The parameter mechanism for externals can be implemented the same as for rules, although that is not likely to happen. Externals do not need a stack frame in most implementations, so their parameters can be kept in, for instance, registers.

3.3.8.1 External rules

ALICE externals are declared to make it easy to implement them as subroutines. In most implementations however in-line code will be generated for them. Standard external rules are identified by both an integer representation and an ALICE symbol. User externals are identified by an integer representation and a string. What is in the string depends on the implementation of user externals. User externals will therefore not appear in portable ALICE programs.

Syntax:

```
extrule decls           : extrule decl, [extrule decls].

extrule decl            : extrule decl symbol, sp, repr, co, stag, el.

stag                    : string;
                          extag.


extag                   : .     .     .
                          subtr symbol;
                          mult symbol;
                          get char symbol;
                          .     .     .
                          .     .     .
                          .     .     .
                          .     .     .
                          put char symbol;
                          put string symbol;
                          get int symbol;
                          put int symbol;
                          get data symbol;
                          put data symbol.

extcall                 : extcall id,
                          [copies to input gate],
                          ext scall or ext fcall,
                          [restores from output gate],
                          extcall end.
extcall id               : ext scall id;
                           ext fcall id.
ext scall id             : ext scall id symbol, sp, repr, co, stag, el.
ext fcall id             : ext fcall id symbol, sp,
                           repr, co, stag, co, false address, el.

ext scall or ext fcall   : ext scall symbol, sp, repr, co, stag, el;
                           ext fcall symbol, sp,
                           repr, co,  stag, co, false address, el.

extcall end              : extcall end symbol, co, repr, el.
```

Semantics:

An extrule declaration macro declares an external represented by the repr parameter of the macro. If the second parameter is a string, the external is a user external with implementation dependent semantics. If the second parameter is an ALICE symbol, the external is a standard external. The semantics of such a standard external are defined in the ALEPH manual. The name of the ALICE symbol is equal to the name of the ALEPH external (e.g. the ALICE "get char symbol" stands for the ALEPH standard external "get char").

A group of extcall macros causes a standard external to be executed with actual parameters as described by the copies to input gate macros and the copies to output gate macros.

The parameters of a scall id and a scall macro have the same meaning as those of a extrule decl macro.

The first two parameters of a fcall id and a fcall macro have the same meaning as those of an extcall decl macro. The third parameter is the false address of the fcall.

The parameter of an extcall end macro is the true address of the extcall.

Installation hints:

hint 1.

The false address and true address of the external call are supplied in the first macro such that, if in-line code instead of a subroutine call is generated, this information is available in time. Gate and stack macros are left out, because the information they carry is already available by means of the ALICE symbol.

hint 2.

Implement the gate in registers. Don't use the run-time stack for an external but let it transform the input gate into the output gate.

hint 3.

A smart ALICE processor could read all macros for an external call before generating any code for it. It could also parse the parameter descriptors to handle simple cases (such as constant parameters) specially.

3.3.8.2 ALICE primitives

ALICE primitives can be divided in two classes. In the first place there are low level operations such as jumping to a label or halting. In the second place there are high level operations such as building a jump table and extending a stack, that cannot be described in terms of an ALICE external, because the parameters of such an operation are of a different nature.

3.3.8.2.1 Low level primitives

Syntax:

```
primitive                 : jump;
                            label definition;
                            source line;
                            exit;
                            class box;
                            class;
                            extension.


jump                      : jump symbol, sp, repr, el.

label definition          : label symbol, sp, repr, el.

source line               : source line symbol, sp,
                            integer, el.

exit                      : exit symbol, sp, repr, co, valref, el.
```

Semantics:

A label macro causes the generation of a label, uniquely represented by the repr parameter of the macro.

A jump macro causes the generation of a jump to the label, represented by the repr parameter of the macro. This label can by generated by means of a label macro, a success tail macro, or a fail tail macro.

A source line macro causes the generation of code, such that the integer parameter of the macro is stored in a memory location, especially allocated for this purpose.

An exit macro causes the generation of code that halts execution and passes the constant, described by the parameters of the macro to the operating system.

Installation hints:

hint 1.
The source line macro serves the run-time system to provide the source line number of the original ALEPH program in error messages. The simplest implementation is to generate a move to a register or memory location. This is of course a time consuming business. Another way is to build from all source line macros a table giving the correspondence between machine addresses and source line numbers.

3.3.8.2.2 Classification primitives

Class box and class macros occur in ALICE as the translation of the ALEPH classifier box respectively the ALEPH areas of a classification.

The ALEPH classifier box is translated to a class box.

The ALEPH areas prefixing the alternatives of the classification are gathered. The alternatives are translated as usual. After the translation of the alternatives the ALEPH areas are translated to a class. A class consists of a class begin macro, a number of zone macros, and a class end macro.

Syntax:

```
class box               : class box id symbol, el,
                          load val in v reg,
                          class box end symbol, sp, representation, el.


class                   : class begin symbol, sp, repr, co,
                          optimize, el,

                          zones,

                          class end symbol, sp, optimize, el.

zones                   : zone bounds, (zones);
                          zone value, (zones).

zone bounds             : zone bounds symbol, sp,
                          minbound, co, maxbound, co,
                          representation, el.

minbound                : repr, co, valref.
maxbound                : repr, co, valref.
```

zone value	: zone value symbol, sp, repr, co, valref, co, representation, el.
optimize	: true symbol; false symbol.

Semantics:

A class box macro has no effect on the ALICE machine. It announces that macros to load the v-register will follow.

A class box end macro causes the generation jump to the label generated by a class begin macro.

A class begin macro causes generation of a label, represented by the repr parameter of the macro. The optimize parameter has the following meaning:

true: the value in the v-register will always fit in one of the zones that follow.

false: it is not guaranteed that the value in the v-register will fit in one of the zones. If not an error handling run-time routine has to be executed.

A zone value macro causes the generation of code to test whether the contents of the v-register equals the constant source referred to in that macro. If the test succeeds flow of control resumes at the address, represented by the repr parameter of the macro.

A zone bounds macro causes the generation of a test whether the contents of the v-register lies between the min bound a the max bound (bounds included). These bounds are constant sources. If the test succeeds flow of control resumes at the address, represented by the repr parameter of the macro.

The optimize parameter of a class end macro has the same meaning as the one of a class begin macro.

Installation hints:

hint 1.

If it is known that the value in the v-register will always fit in one of the zones, efficient code can be generated such that few tests really have to be done. Consider for example a class with three zone bounds macros with the following bounds and reprs

min bound	max bound	repr
-32	128	121
0	256	121
512	1024	133

The ALICE processor could generate the following code if optimization is allowed:

```
if v-reg < 257  goto 1121 else goto 1133
```

The ALICE processor is allowed to generate this kind of optimized code if the optimize parameter in the class begin and class end macro is true. If the optimize parameter in these macros is false, a (jump to an) error routine must be generated at the end of the tests. In that case from the same three zone bounds macros as above the following code could be generated:

```
if (v-reg > -33 and v-reg < 257) goto 1121
else if (v-reg > 511 and v-reg < 1025) goto 1133
     else goto classerror
```

3.3.8.2.3 Extension primitives

Extension macros are the translation of an ALEPH extension. The sources are put on the gate as input parameters. The a-register is loaded with the address of the administration of the stack that must be extended. Then the administration of the stack can be updated. The formals in the extension copy macros indicate the position of the new elements.

Syntax:

```
extension                : extension id,
                           input gate creation,
                           copies to input gate,
                           load addr in a reg,
                           extension call,
                           extension copies,
                           extension end.

extension id             : extension id symbol, el.

extension call           : extension call symbol, el.

extension copies         : extension copy, [extension copies].

extension copy           : extension copy symbol, sp,
                           integer, co, integer, el.

extension end            : extension end symbol, sp, representation, el.
```

Semantics:

An extension id macro has no effect on the ALICE abstract machine. It announces that the gate will be filled with sources to extend a list.

An extension call macro causes the generation of code for the initiation of an extension: room is made to push a new block on the list. The gate contains the sources to extend the list with. The a-register contains the address of the administration of the list that will be extended.

An extension copy macro causes the copying of a gate element to a position in the new block of the list. The first integer parameter of the macro denotes the position on the gate. The second integer parameter indicates which element of the block has to be selected just as the integer parameter of a loadv list elem macro.

The repr parameter of an extension end macro is the representation of the true address of the extension.

Installation hint:

hint 1.
An extension consists of:
1. Stacking the sources on the gate.
2. Ensuring the extension.
3. Updating the list administration.
4. Transporting the sources to the selected positions.

ad 2. If not all virtual addresses are mapped in the physical address space, check whether the extension is still possible in the physical address space. If not, re-allot the lists or swap in a new page, depending on the list allocation.

ad 3. Update the right limit and implementation dependent core limits, if any.

ad 4. The gate is not used stackwise here. One gate element can be copied to more stack positions.

3.4. Program structure

This section is concerned with macros other than value, data, and rule macros. Their purpose is to provide global information and to separate groups of macros, such as data and rule macros, from each other.

Syntax:

```
ALICE program           : program id symbol, sp, string, el,
                          status information,
                          values option,
                          end values symbol, el,
                          data,
                          communication area,
                          rules,
                          end symbol, sp, string, el.

status information      : status symbol, sp,
                          integer, co,
                          integer, co,
                          integer, co,
                          integer, co,
                          integer, co,
                          integer, co,
                          integer, co,
                          integer, co,
                          integer, el.

communication area      : communication symbol, sp,
                          repr, co,
                          repr, co,
                          repr, co,
                          string, el,

                          status information.


rules                   : extrule decls,
                          rules and root.


rules and root          : [rule decls], root, [rule decls].

root                    : root symbol, sp, string, el,
                          affix form,
                          exit.

affix form              : call;
                          extcall.
```

Semantics:

The parameters of a status macro have the following meaning:
First parameter, integer: maximal number of parameters of a call (formals + locals).
Second parameter, integer: maximal number of input or output parameters to be put on the gate.
Third parameter, integer: number of locations needed in the value table.
Fourth parameter, integer: number of variables to be declared.
Fifth parameter, integer: number of files to be declared.
Sixth parameter, integer: number of breathing lists to be declared.
Seventh parameter, integer: number of non-breathing lists to be declared.
Eighth parameter, integer: background:
 0: the original ALEPH program contains no background pragmats.
 1: the original ALEPH program contains background pragmats.
Ninth parameter, integer: dump:
 0: no dump is requested
 1: a rule dump is requested
 2: a global dump is requested
 3: a member dump is requested

The meaning of the dump parameter is explained in detail in the ALEPH manual (dump pragmat).

The string parameters of the program id macro, the communication macro, the root macro, and the end macro of an ALICE program are identical. They contain the "title" of the program.

The parameters of the communication macro have the following meaning:
First parameter, repr: representation of the first list in the chain of list administrations.
Second parameter, repr: representation of the first file in the chain of file administrations.
Third parameter, repr: representation of the first variable in the chain of variable declarations.
Fourth parameter, string: the title of the program.

The execution of an ALICE program starts at the affix form (call or extcall) following the root macro.

Installation hints:

hint 1.

The title string parameter in the root macro can serve to generate an entry point. The first thing to do in most implementations will be initialization of file administrations and maybe of list administrations. Code for this can be generated from the communication macro. In that case a call to that code will be generated from the root macro.

hint 2.

The status macro informs the ALICE processor what kind of ALICE program it is going to have to translate. If, for example, parameters are passed in registers and the maximal gate is greater than the number of registers available for parameter passing, the ALICE processor is warned in advance so that it can take appropriate action.

3.5 ALICE grammar

```
$
 ALICE grammar october 77
 ALICE consists of a set of macros,
 comment lines, and pragmat lines.

 A comment line starts with "xxx " and should be ignored.

 A pragmat line has the format:

 pragmat symbol,  sp, string, el.

 The string will be passed on to the assembler

 A portable program contains no pragmat lines

 A macro has the form:

 macro                  : mac name, [sp, parameters], el.
 mac name               : ALICE terminal symbol.
 parameters             : parameter, [co, parameters].
 parameter              : string;
                          integer;
                          character;
                          ALICE terminal symbol.

$
```

```
$ ALICE terminal symbols       representation   $


$ macro names $
add symbol;                         $      add      $
begin file adm symbol;              $      bfa      $
call id symbol;                     $      cll      $
class begin symbol;                 $      csb      $
class end symbol;                   $      cse      $
char denotation symbol;             $      chd      $
constant source symbol;             $      css      $
communication symbol;               $      cmm      $
copy a reg symbol;                  $      car      $
copy from input gate symbol;        $      cig      $
copy v reg symbol;                  $      cvr      $
divide symbol;                      $      dvd      $
end file adm symbol;                $      efa      $
end list symbol;                    $      els      $
end symbol;                         $      end      $
end values symbol;                  $      eva      $
exit symbol;                        $      ext      $
ext fcall symbol;                   $      efc      $
ext scall symbol;                   $      esc      $
ext table length symbol;            $      etl      $
ext table decl symbol;              $      etd      $
extcall end symbol;                 $      ece      $
ext scall id symbol;                $      esi      $
ext fcall id symbol;                $      efi      $
extension call symbol;              $      etc      $
extension copy symbol;              $      exc      $
extension end symbol;               $      exe      $
extension id symbol;                $      exi      $
extrule decl symbol;                $      erl      $
fail tail id symbol;                $      fti      $
fallow symbol;                      $      flw      $
fcall symbol;                       $      fcl      $
class box id symbol;                $      cbi      $
end class box symbol;               $      ebx      $
free w reg symbol;                  $      frw      $
indexed input parameter symbol;     $      iip      $
indexed output parameter symbol;    $      iop      $
input gate symbol;                  $      igt      $
int symbol;                         $      int      $
int fill symbol;                    $      itf      $
jump symbol;                        $      jmp      $
label symbol;                       $      lab      $
link symbol;                        $      lnk      $
list adm symbol;                    $      ldm      $
list symbol;                        $      lst      $
loada global symbol;                $      lag      $
loada stack var symbol;             $      las      $
loadv constant symbol;              $      lvc      $
loadv limit symbol;                 $      lvl      $
loadv list elem symbol;             $      lvi      $
loadv stack var symbol;             $      lvs      $
loadv variable symbol;              $      lvv      $
```

```
loadw symbol;                     $      ldw     $
manifest constant symbol;         $      mcn     $
multiply symbol;                  $      mul     $
rule id symbol;                   $      rli     $
numerical symbol;                 $      num     $
output gate symbol;               $      ogt     $
pointer symbol;                   $      ptr     $
program id symbol;                $      pid     $
restore to output gate symbol;    $      rog     $
root symbol;                      $      rut     $
source line symbol;               $      srl     $
scall symbol;                     $      scl     $
stack frame symbol;               $      sfr     $
status symbol;                    $      sts     $
storew variable symbol;           $      swv     $
storew list element symbol;       $      swi     $
storew stack var symbol;          $      sws     $
string length symbol;             $      sln     $
string fill symbol;               $      str     $
subtract symbol;                  $      sub     $
success tail id symbol;           $      sti     $
target stack frame symbol;        $      tsf     $
unstack and return symbol;        $      unr     $
variable symbol;                  $      var     $
zone bounds symbol;               $      znb     $
zone value symbol;                $      znv     $

$ delimiters $
space symbol;                     $      " "     $
comma symbol;                     $       ,      $
end of line;                      $ medium dependent $

$ parameters $
new line symbol;                  $      nln     $
same line symbol;                 $      sln     $
rest line symbol;                 $      rln     $
new page symbol;                  $      npg     $
max char symbol;                  $      mxc     $
word size symbol;                 $      wsz     $
max int symbol;                   $      mxi     $
min int symbol;                   $      mni     $
int size symbol;                  $      isz     $
comma-tag symbol;                 $      com     $
space-tag symbol;                 $      spc     $
min addr symbol;                  $      mna     $
max addr symbol;                  $      mxa     $
virt length symbol;               $      vln     $
nil symbol;                       $      nil     $
false symbol;                     $      fls     $
true symbol;                      $      tru     $

add-tag symbol;                   $      add     $
subtr symbol;                     $      sub     $
mult symbol;                      $      mul     $
divrem symbol;                    $      div     $
plus symbol;                      $      pls     $
minus symbol;                     $      min     $
```

```
times symbol;                     $     tms     $
incr symbol;                      $     inc     $
decr symbol;                      $     dec     $
less symbol;                      $     les     $
lseq symbol;                      $     lsq     $
more symbol;                      $     mor     $
mreq symbol;                      $     mrq     $
equal symbol;                     $     eql     $
noteq symbol;                     $     ntq     $
random symbol;                    $     rnd     $
set random symbol;                $     srn     $
set real random symbol;           $     srr     $
sqrt symbol;                      $     sqr     $
pack int symbol;                  $     pki     $
unpack int symbol;                $     upi     $
bool invert symbol;               $     biv     $
bool and symbol;                  $     bnd     $
bool or symbol;                   $     bor     $
bool xor symbol;                  $     xor     $
left circ symbol;                 $     lci     $
right circ symbol;                $     rci     $
right clear symbol;               $     rcl     $
is elem symbol;                   $     isl     $
is true symbol;                   $     itr     $
is false symbol;                  $     isf     $
set elem symbol;                  $     sel     $
clear elem symbol;                $     cll     $
extract bits symbol;              $     exb     $
first true symbol;                $     ftr     $
pack bool symbol;                 $     pkb     $
unpack bool symbol;               $     upb     $
to ascii symbol;                  $     tsc     $
from ascii symbol;                $     fsc     $
pack string symbol;               $     pks     $
unpack string symbol;             $     ups     $
string elem symbol;               $     ste     $
string length-tag symbol;         $     stl     $
compare string symbol;            $     cms     $
unstack string symbol;            $     uns     $
previous string symbol;           $     pvs     $
was symbol;                       $     was     $
next symbol;                      $     nxt     $
previous symbol;                  $     prv     $
list length symbol;               $     lsl     $
unstack symbol;                   $     utk     $
unstack to symbol;                $     ust     $
unqueue symbol;                   $     unq     $
scratch symbol;                   $     scr     $
get line symbol;                  $     gln     $
put line symbol;                  $     pln     $
get char symbol;                  $     gch     $
put char symbol;                  $     pch     $
put string symbol;                $     pst     $
get int symbol;                   $     gnt     $
put int symbol;                   $     pnt     $
get data symbol;                  $     gdt     $
put data symbol;                  $     pdt     $
```

```
$ other primitives $
string;                           $ character sequence
                                  delimited by quotes
                                  quotes in the string are represented by
                                  quote images [""]
                                  $

character;

integer.                          $ unsigned digit sequence $
```

```
ALICE program          : program id symbol, sp, string, el,

                         status information,

                         values,
                         end values symbol, el,

                         data,

                         communication area,

                         rules,

                         end symbol, sp, string, el.


data                   : [constant sources],
                         [variable decls],
                         [lists],
                         [files].


lists                  : [list areas],
                         [external table decls],
                         [list administrations].

files                  : file administrations.

rules                  : extrule decls,
                         rules and root.



sp                     : space symbol.
co                     : comma symbol.
el                     : end of line.



status information     : status symbol, sp,
                         integer, co, $ maximal stack frame $
                         integer, co, $ maximal gate size $
                         integer, co, $ number of expressions $
                         integer, co, $ number of variables $
                         integer, co, $ number of files $
                         integer, co, $ number of breathing lists $
                         integer, co, $ number of non-breathing lists $
                         integer, co, $ background:
                                      0: No lists on background
                                      1: Lists on background $
                         integer, el. $ dump:
                                      0: no dump
                                      1: rule dump
                                      2: global dump
                                      4: member dump $
```

```
values                  : value, [values].

value                   : value definition;
                          calculation.

value definition        : int denotation;
                          manifest constant;
                          char denotation;
                          string length;
                          external table length.


int denotation          : int symbol, sp, location, co, integer, el.

manifest constant       : manifest constant symbol, sp,
                          location, co, manco, el.

manco                   : new line symbol;
                          same line symbol;
                          rest line symbol;
                          new page symbol;
                          max char symbol;
                          word size symbol;
                          max int symbol;
                          min int symbol;
                          int size symbol;
                          comma-tag symbol;
                          space-tag symbol;
                          min addr symbol;
                          max addr symbol;
                          virt length symbol;
                          nil symbol.

char denotation         : char denotation symbol, sp, location, co,
                          character, el.

string length           : string length symbol, sp,
                          location, co, integer, el.



external table length   : ext table length symbol, sp, location,
                          co, string, el.

$ The string is an exact copy of the ALEPH string including
  the surrounding quotes $



calculation             : operator, sp, location, co,
                          valref, co, valref, el.

operator                : add symbol;
                          subtract symbol;
                          multiply symbol;
                          divide symbol.
```

```
location                : integer.
$ This integer denotes where to put a certain value in the
  table the ALICE processor is building.
  The location will be referred to by valrefs $

valref                  : integer.
$ A valref references the location of an already defined
  value in the table the ALICE processor is building up $

constant sources        : constant source,
                          [constant sources].

constant source         : constant source symbol, sp, repr, co, valref, el.


repr                    : integer.

$ a repr either represents an ALICE object uniquely (>0)
  or it indicates the non-presence of an ALICE object (=0)
$


list areas              : list area,
                          [list areas].

list area               : list symbol, sp, repr, co,    $ of list $
                          list type, co,
                          valref, el,   $ number of virtual addresses $
                          [list fillings],
                          end list symbol, sp, repr, co,
                          list type, co, valref, el.

                          $ The parameters of end list are the same as those
                          of list $

list fillings           : list filling, [list fillings].
list filling            : int fill symbol, sp, valref, el;
                          string fill symbol, sp, string, el;
                          fallow symbol, sp, valref, el.

                          $ fallow stands for uninitialized space
                          to be grabbed for a stack
                          with an absolute size estimate $

variable decls          : variable decl, [variable decls].

variable decl           : variable symbol, sp, repr, co, valref, co,
                          repr, co, $ of next variable $
                          string, el. $ ALEPH variable tag in quotes $
```

```
file administrations     : file administration, [file administrations].

file administration      : begin file adm symbol, sp, file info, el,
                           [pointer area],
                           [numerical area],
                           end file adm symbol, sp, file info, el.

file info                : repr, co,
                           file type, co,
                           $ an integer:
                            0: scratch charfile
                            1: scratch datafile
                            2: input charfile
                            3: input charfile
                            4: output charfile
                            5: output datafile
                            6: input-output charfile
                            7: input-output datafile
                           $

                           repr, co,      $ next file administration $
                           string.        $ file name$

file type                : integer, co, integer.




numerical area           : numerical symbol, sp,
                           valref, co,    $ lower bound$
                           valref, el,    $ upper bound$
                           [numerical area].

pointer area             : pointer symbol, sp, repr, el, $ of a list $
                           [pointer area].
```

```
list administrations     : list administration, [list administrations].

list administration      : list adm symbol, sp,
                           list info, el.


list info                : repr, co,      $ of the list$
                           list type, co,
                           valref, co,    $ virtual min  $
                           valref, co,    $ virtual max  $
                           valref, co,    $ virtual left $
                           valref, co,    $ virtual right $
                           valref, co,    $ calibre      $
                           repr, co,      $ next list adm $
                           string.        $ name of the list$

list type                : integer.

$ 0: not breathing, no background pragmat
  1: not breathing,    background pragmat
  2:     breathing, no background pragmat
  3:     breathing,    background pragmat
$

external table decls     : external table decl, [external table decls].

external table decl      : ext table decl symbol, sp,
                           list info, co,
                           string, el. $ ALEPH string $



communication area       : communication symbol, sp,
                           repr, co,  $ first list $
                           repr, co,  $ first file $
                           repr, co,  $ first variable $
                           string, el,

                           status information.
```

```
extrule decls          : extrule decl, [extrule decls].


extrule decl           : extrule decl symbol, sp, repr, co, stag, el.

stag                   : string;
                         extag.


$ If the external is a standard external, the stag is an extag;
 The externals of a portable program must be standard externals $

extag                  : add-tag symbol;
                         subtr symbol;
                         mult symbol;
                         divrem symbol;
                         plus symbol;
                         minus symbol;
                         times symbol;
                         incr symbol;
                         decr symbol;
                         less symbol;
                         lseq symbol;
                         more symbol;
                         mreq symbol;
                         equal symbol;
                         noteq symbol;
                         random symbol;
                         set random symbol;
                         set real random symbol;
                         sqrt symbol;
                         pack int symbol;
                         unpack int symbol;
                         bool invert symbol;
                         bool and symbol;
                         bool or symbol;
                         bool xor symbol;
                         left circ symbol;
                         right circ symbol;
                         right clear symbol;
                         is elem symbol;
                         is true symbol;
                         is false symbol;
                         set elem symbol;
                         clear elem symbol;
                         extract bits symbol;
                         first true symbol;
                         pack bool symbol;
                         unpack bool symbol;
                         to ascii symbol;
                         from ascii symbol;
                         pack string symbol;
                         unpack string symbol;
                         string elem symbol;
                         string length-tag symbol;
                         compare string symbol;
                         unstack string symbol;
```

```
                          previous string symbol;
                          was symbol;
                          next symbol;
                          previous symbol;
                          list length symbol;
                          unstack symbol;
                          unstack to symbol;
                          unqueue symbol;
                          scratch symbol;
                          get line symbol;
                          put line symbol;
                          get char symbol;
                          put char symbol;
                          put string symbol;
                          get int symbol;
                          put int symbol;
                          get data symbol;
                          put data symbol.




rules and root          : [rule decls], root, [rule decls].


rule decls              : rule decl, [rule decls].

root                    : root symbol, sp, string, el,         $ title $
                          source line,
                          affix form,
                          exit.


affix form              : call;
                          extcall.

rule decl               : rule head, rule body, rule tail.
rule head               : rule id,
                          stack frame,
                          [copies from input gate].


rule id                 : rule id symbol, sp, repr, co, rule type, co,
                          recursion, co, string, el.


rule type               : integer.
                          $ 0: cannot fail
                            1: can fail
                          $

recursion               : integer.
                          $ 0: not recursive
                            1: recursive
                          $
```

```
stack frame             : stack frame symbol, sp, number of parameters,
                          co, number of locals, el.


rule tail               : success tail,
                          [fail tail].

success tail            : success tail id,
                          [output gate creation],
                          [restores to output gate],
                          unstack and return true.

success tail id         : success tail id symbol, sp, repr, co,
                          rule type, co, recursion, el.

output gate creation    : output gate symbol, sp, size of output gate, el.
size of output gate     : integer.

unstack and return true : unstack and return symbol, sp,
                          number of parameters, co, number of locals, co,
                          true symbol, el.


fail tail               : fail tail id,
                          unstack and return false.

fail tail id            : fail tail id symbol, sp, repr, co,
                          rule type, co,
                          recursion, el.

unstack and return false: unstack and return symbol, sp,
                          number of parameters, co, number of locals, co,
                          false symbol, el.


copies from input gate  : copy from input gate, [copies from input gate].
copy from input gate    : copy from input gate symbol, sp, formal, el.
formal                  : position on gate, co, position on stack.
position on gate        : integer.
position on stack       : integer.

restores to output gate : restore to output gate,
                          [restores to output gate].
restore to output gate  : restore to output gate symbol, sp, formal, el.


rule body               : statements.
statements              : statement, [statements].
statement               : call;
                          extcall;
                          primitive.
```

```
primitive               : jump;
                          source line;
                          class box;
                          class;
                          extension;
                          exit;
                          label definition.

call                    : call id,
                          input gate creation,
                          target stack frame,
                          [copies to input gate],
                          scall or fcall,
                          [restores from output gate],
                          link.


call id                 : call id symbol, sp, repr, co,
                          rule type, co, recursion, el.

input gate creation     : input gate symbol, sp, size of input gate, el.
size of input gate      : integer.

target stack frame      : target stack frame symbol, sp,
                          number of parameters, co,
                          number of locals, el.
number of parameters    : integer.
number of locals        : integer.


scall or fcall          : scall symbol, sp, repr, el;

                          fcall symbol, sp, repr, co,
                          false address, el.


false address           : repr.
                          $ of a label $

link                    : link symbol, sp, true address, el.

true address            : repr.

extcall                 : extcall id,
                          [copies to input gate],
                          ext scall or ext fcall,
                          [restores from output gate],
                          extcall end.

extcall id              : ext scall id;
                          ext fcall id.
ext scall id            : ext scall id symbol, sp, repr, co, stag, el.

ext fcall id            : ext fcall id symbol, sp,
                          repr, co, stag, co, false address, el.
```

```
ext scall or ext fcall   : ext scall symbol, sp, repr, co, stag, el;
                           ext fcall symbol, sp,
                           repr, co,  stag, co, false address, el.

extcall end              : extcall end symbol, co, true address, el.


jump                     : jump symbol, sp, repr, el.

source line              : source line symbol, sp, line number, el.
line number              : integer.

class box                : class box id symbol, el,
                           load val in v reg,
                           end class box symbol, sp, true address, el.
                           $ true address = repr of class $



class                    : class begin symbol, sp, repr, co,
                           optimize, el,
                           zones,
                           class end symbol, sp, optimize, el.
zones                    : zone bounds, [zones];
                           zone value,  [zones].

zone bounds              : zone bounds symbol, sp,
                           minbound, co, maxbound, co,
                           true address, el.

minbound                 : repr, co, valref.
maxbound                 : repr, co, valref.

zone value               : zone value symbol, sp, repr, co, valref, co,
                           true address, el.

optimize                 : true symbol;
                           false symbol.


extension                : extension id,
                           input gate creation,
                           copies to input gate,
                           load addr in a reg,  $ stack adm $
                           extension call,
                           extension copies,
                           extension end.

extension id             : extension id symbol, el.

extension call           : extension call symbol, el.

extension copies         : extension copy, [extension copies].
extension copy           : extension copy symbol, sp, formal, el.

extension end            : extension end symbol, sp, true address, el.
```

```
exit                      : exit symbol, sp, valref, el.

label definition          : label symbol, sp, repr, el.



copies to input gate      : copy to input gate,
                            [copies to input gate].

copy to input gate        : copy val to input gate;
                            copy addr to input gate.

copy val to input gate    : load val in v reg,
                            copy v reg to input gate.

load val in v reg         : load simple in v reg;
                            load indexed element in v reg.

load simple in v reg      : load constant in v reg;
                            load variable in v reg;
                            load limit in v reg;
                            load stack var in v reg.


copy v reg to input gate: copy v reg symbol, sp, formal, el.

copy addr to input gate : load addr in a reg,
                          copy a reg to input gate.

load addr in a reg        : load global addr in a reg;
                            load stack var in a reg.

copy a reg to input gate: copy a reg symbol, sp, formal, el.




load constant in v reg   : loadv constant symbol, sp, repr, co, valref, el.

load variable in v reg   : loadv variable symbol, sp, repr, el.

load limit in v reg      : load addr in a reg,
                           loadv limit symbol, sp, limit type, el.
limit type               : integer.
                           $
                           0: left
                           1: right
                           2: calibre
                           $

load stack var in v reg : loadv stack var symbol, sp,
                          position on stack, el.
```

```
load indexed element
        in v reg       : indexed input parameter symbol, el,
                         load simple in v reg,
                         load list element in v reg sequence.

load list element in
        v reg sequence  : load list element in v reg,
                         [load list element in v reg sequence].


load list element
        in v reg       : load addr in a reg,
                         loadv list elem symbol, sp, integer, el.
                         $ 0: right most element of indexed block

                          i: (i-1)-th right most element $



load global addr
        in a reg       : loada global symbol, sp, repr, el.

load stack var
        in a reg       : loada stack var symbol, sp,
                         position on stack, el.



restores from
        output gate    : restore from output gate,
                         [restores from output gate].

restore from output gate: copy gate elem to w reg,
                         store w reg sequence,
                         free w reg.
```

```
store w reg sequence    : store w reg, [store w reg sequence].

copy gate elem to w reg : loadw symbol, sp, formal, el.

store w reg             : store w reg in variable;
                          store w reg in list element;
                          store w reg in stack var.

store w reg in variable : storew variable symbol, sp, repr, el.

store w reg in
        list element    : indexed output parameter symbol, el,
                          load simple in v reg,
                          [load list element in v reg sequence],
                          load addr in a reg,
                          storew list element symbol, sp, integer, el.

store w reg in
        stack var       : storew stack var symbol, sp,
                          position on stack, el.


free w reg              : free w reg symbol, el.
```

4 References

1 D. Grune, R. Bosch & L.G.L.T. Meertens
ALEPH Manual, Report IW 17/75
Mathematisch Centrum Amsterdam 1975

2 Waite, W. M.
Hints on distributing Portable Software
Software - Practice and Experience, vol. 5, 295-308 (1975)

3 P.J. Brown
ML/1 user's manual (fourth edition)
University of Kent at Canterbury, 1970

4 Waite, W.M.
The mobile programming system: STAGE2
Comm. ACM 13,7, pp. 415-421 (1973)

5 W.J. Hansen & H. Boom
The report on the standard hardware representation for ALGOL 68
SIGPLAN Notices, Vol 12, Number 5, 80-87 (1977)

6 Poole, P.C. & Waite, W.M.
Portability and Adaptability, <u>in</u>
F.L. Bauer (Ed.), Advanced course on Software Engineering,
Springer-Verlag, Berlin, 183-277 (1973)

7 Waite, W.M.
Implementing Software for non-numeric applications
Prentice-Hall, Englewood Cliffs, N.J., 1973

8 P.J. Brown
Macro processors and techniques for Portable Software
John Wiley 1974

9 P.J. Brown (Ed.)
Software Portability An advanced Course
Cambridge University Press 1977

10 A.S. Tanenbaum, P. Klint & W. Bohm
Guidelines for Program Portability, Report IW 88/77
Mathematisch Centrum Amsterdam 1977