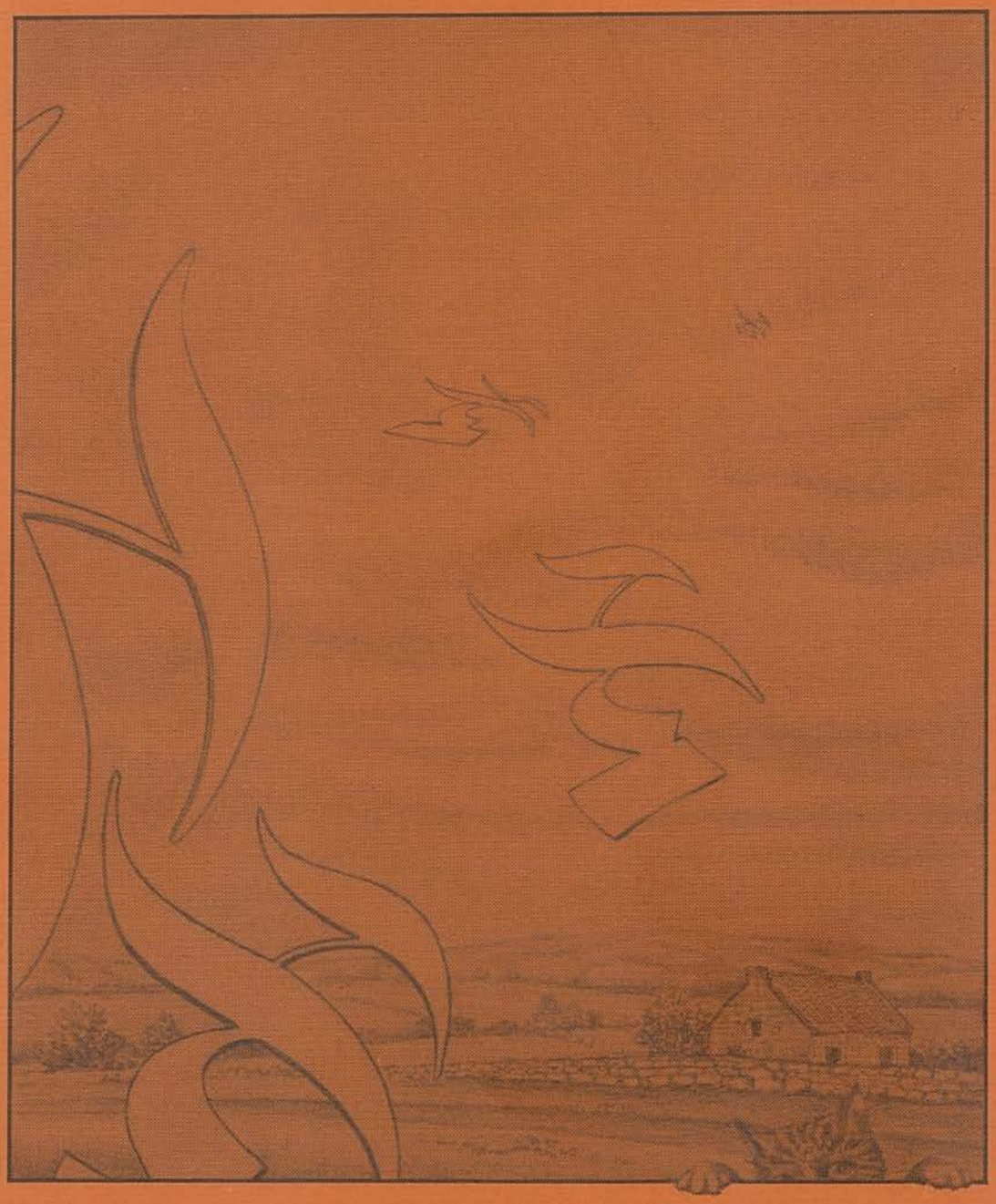

On the Design of □ ALEPH □

Dick Grune

On the Design of □ ALEPH □

On the Design of □ ALEPH □

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor in de Wiskunde en Natuurwetenschappen aan de Universiteit van Amsterdam, op gezag van de Rector Magnificus, Dr. D.W. Bresters, hoogleraar in de Faculteit der Wiskunde en Natuurwetenschappen, in het openbaar te verdedigen in de Aula der Universiteit (tijdelijk in de Lutherse Kerk, ingang Singel 411, hoek Spui) op woensdag 15 september 1982 des namiddags te 3 uur precies

door

Dick Grune

geboren te Enschede

Mathematisch Centrum

1982

Promotor: Prof. Dr. Ir. A. van Wijngaarden
Coreferent: Prof. C.H.A. Koster

Bij de voorplaat:

Zomin als de voorplaat de hele Aleph kon omvatten, kon dit proefschrift het hele ALEPH*-project omvatten.*

aan Oom Piet

On the Design of ALEPH

The ALEPH Manual

Current address of the author:
Vakgroep Informatica,
Wiskundig Seminarium,
Vrije Universiteit,
De Boelelaan 1081,
1081 HV Amsterdam.

1. INTRODUCTION

1.1. The language*

ALEPH (*A Language Encouraging Program Hierarchy*) is a high-level programming language designed to induce the user to write his programs in a well-structured way. The language is particularly suitable for problems that suggest top-down analysis (parsers, search algorithms, combinatorial problems, artificial intelligence problems etc.).

An ALEPH procedure is a top-down description of what is to be done: complex actions are defined in terms of (usually) less complex ones, which in turn are defined in terms of still simpler ones, and so on, until a level is reached at which further decomposition is undesirable.

An ALEPH program consists of a set of such definitions, in a notation not unlike the rules of an affix grammar [KOSTER 71b, MEIJER 80]. In fact, many of the ideas in ALEPH were derived from the theory of affix grammars; for example, repetition is expressed not by a GOTO or WHILE statement but by what in a grammar would be called 'right recursion' [BOSCH, GRUNE & MEERTENS 73, GRUNE 75].

The syntax and semantics of ALEPH are so simple that it is possible to derive statically various interesting properties of the dynamic behaviour of the program. For example, the compiler can easily verify that no variable will be used before it has obtained a value. Thus the use of uninitialized variables is prevented in a natural way, without resorting to the (dangerous) trick of automatic initialization. Also, it is possible to detect statically anomalies in the program structure corresponding to the need for "backtrack" in parsing, and provide a message. The signalling of such side effects turns out to be a powerful weapon against messy programming.

The semantic simplicity of ALEPH, especially of its parameter mechanism, easily leads to efficient object code, even without using fancy optimizing techniques. The programmer can formulate his algorithms with all the elegance inherent in a top-down formulation, and still obtain good machine code [WICHMANN 77, BÖHM 78].

Because the semantic primitives needed for the translation are small in number and simple in nature ('pass parameter', 'call procedure conditionally', etc.), the transfer of the compiler from one machine to another is quite straightforward. As, however, additional semantic primitives may be defined by the programmer (e.g., multilength arithmetic, 'convert to hash code', or whatever he thinks is a primitive of his problem), the portability of the program (as opposed to that of the compiler) is determined by the portability of these programmer-defined primitives.

The work presented here is a continuation of the research started by C.H.A. Koster, which resulted in the development of CDL (*Compiler Description Language*) [KOSTER 74]. His CDL-compiler gave us a great deal of experience with affix-grammar-like languages, from which ALEPH has benefited.

---

*This section is an abridged version of [BOSCH, GRUNE & MEERTENS 73].

1.1.1. Goals

Our main goals in the design of ALEPH were the following:

a. ALEPH must allow good programming at a reasonable effort and a moderate price.
b. Since ALEPH is a tool and not a goal in itself, the compiler for it must be simple.
c. To allow the application of the algorithms written in ALEPH on a wide range of machines, the compiler must be portable (as far as possible).

The above requirements were augmented by two more requirements of a more practical nature:

d. Since in our institute ALEPH is mainly intended for compiler writing, sorting algorithms, text-editing, etc., emphasis is on facilitating non-numeric programming.
e. Since the project had to be executed on early and mid third generation computing equipment, the compiler must not require any advanced hardware.

Sub a.

Two different approaches were made for the effecting of such a vague notion as 'good programming'. First, the literature contains ideas about what constitutes good programming [DAHL, DIJKSTRA & HOARE 72, DIJKSTRA 76, LINGER, MILLS & WITT 79]; many of these ideas were incorporated. Second, we often found it much easier to recognize bad programming and forbid it than to recognize good programming and to promote it.

Our most powerful weapon against bad programming is the 'static semantic check', applicable in those situations in which the structure of the language allows the compiler to check *statically* (i.e., during compilation) whether the *semantics* makes sense (during run time). Examples are: mode checking in ALGOL 68, which detects the (nonsensical) storing of a value of one type under a name of a different type; or, more primitively, the block structure in many high-level languages which detects the (nonsensical) access to a dynamically non-existing item. ALEPH should amply allow such tests.

It is of course not possible to disallow bad programming in general: a language powerful enough to formulate any algorithm in it is also powerful enough to formulate it messily. Nevertheless, it is often possible to make the 'desirable' construction more convenient than an 'undesirable' one: the way a language is used does not so much depend on its possibilities (it is a Turing machine anyway) as on the convenience of those possibilities. Although it is perfectly possible to write recursive routines in FORTRAN, hardly anybody ever does so, as the administration is just too cumbersome, and, conversely but analogously, it is perfectly possible to 'jump all over the place' in ALEPH but hardly anybody ever does so, as the administration is just too cumbersome.

We require 'good programming' to be available 'at a reasonable effort'. Consequently, if a feature normally present and useful in programming languages is banished from ALEPH, an acceptable alternative should be present.

We also require 'good programming' 'at a moderate price'. Since the only way to program a machine efficiently is in hard machine code, we should be willing to accept certain losses in writing in a high-level language. These losses, however, must not depend on the style of programming in such a way as to foster bad programming: for example, in many high-level languages it is more efficient to pass information to

procedures in global variables than in parameters. Consequently, ALEPH should allow efficient implementation of those features we consider to lead to good programming.

Sub b.

The required simplicity of the compiler conflicts with the tendency to make ALEPH as high-level as possible and with the need for extensive static checking. Some trade-off is to be expected here.

Sub c.

The greatest portability problem in compiler construction is the portability of the object code. Traditionally, compilers are written for one specific language and for one specific machine. Converting such a compiler to a different machine is often nearly impossible due to fundamental differences in the object code. We shall have to make a conscious effort to restrict these conversion problems to a bearable minimum, or, better still, to avoid conversion at all.

Sub e.

Fancy hardware like virtual memory, hardware stack or microprogramming is not supposed to be available. Consequently, some fairly elaborate analyses like check on non-recursivity are worth while. Nevertheless the object code could still make good usage of the above advanced features.

1.1.2. Realization

Sub a and b.

A good basis for the design of our programming language was found in the concept of a 'formal grammar'. Normally a formal grammar is used to describe the admissible programs in the language being defined, but that is not the application we have in mind here. Just as we can use a grammar to produce (program) texts, we can use a grammar to produce directly the solutions to our problem. Since we want the solutions to be produced mechanically, we are forced to consider the grammar as a program, and write a producer (interpreter or compiler) for it. Investigation in this area causes the borders between grammars and programs to fade away. A. van Wijngaarden has given an application of this idea in its purest form [VAN WIJNGAARDEN 81].

The process of converting this abstract idea into a practical, efficient programming language is described in section 3.3. The syntactic and semantic simplicity of formal grammars (as compared to those of programs) have had important consequences for ALEPH: aspects of the dynamic behaviour of an ALEPH program can be derived statically and used in a static semantic check; straightforward implementation is already quite efficient; and machine-independence is high.

Sub c.

Our solution to the problem of the object code portability is to produce machine-independent intermediate code of a very simple nature, ALICE [BÖHM 77] (section 4.4 in this thesis). This code can be produced internally and converted directly to pertinent machine code (for production) or it can be produced externally and then be converted separately by a simple ad-hoc program.

Sub e.

In the absence of advanced memory hardware, measures must be taken to make efficient data storage available in a convenient way. We have found a good solution in 'extensible arrays' with unique indices. This facility is described in 3.5.1 and in

paragraph 4.1.4 of the ALEPH Manual.

1.2. The compiler

Much has been written about specific topics in compiler construction. For parsing one has an ample choice of methods, all well described: top-down [KNUTH 71], bottom-up [DEREMER 71], operator-precedence [FLOYD 63] and many others; a comprehensive account is given in [LEWIS II, ROSENKRANTZ & STEARNS 76]. Likewise, code generation is widely studied, though perhaps less extensively and more ad-hoc than parsing: tree-walking, common sub-expressions, intermediate codes, threaded code and peep-hole optimization, to mention a few subjects. Since these subjects are not often treated in isolation it is more appropriate here to refer to general works like [AHO & ULLMAN 78], [WULF et al. 75] or [BAUER & EICKEL 74].

All these studies provide specific algorithms to be plugged in in a general framework considered given (or trivial). Hardly any attention is given to the question of how such a framework should be designed or even why it should look the way it was given. The data flow inside the compiler (not to be confused with the data flow inside the translated program!) is largely ignored.

Since the design of the new ALEPH compiler was a one-person project, I needed a firm technique to guide me in designing the framework and the information flow in it. The technique I have used can be best described as 'demand-driven'. Faced with a well-defined source language to start from, viz. ALEPH, and a well-defined target code to aim at, viz. ALICE, we are tempted to start a classical design process from ALEPH to ALICE to bridge the gap. The disadvantage is that the steps in this process are largely arbitrary, given by intuition or tradition. Especially in the beginning it is not at all clear what information in the source text should receive attention. Examples are: 'Should comments be kept?', 'Where does the **program-title** go?' or 'Do we have to keep track of the largest number of parameters ever used in a procedure call?'.

If, however, we start from the target code, it is immediately clear from its specifications what information is demanded by each of its instructions. These demands then give rise to other demands, which, by working backwards, we can hope to fulfil eventually from the source code. By applying this technique in its purest form, we would, in the end, be faced with the demand for a 'parsing' of the source code.

The design technique is described in more detail in section 4.2. Part of its results are shown in chapter 5.

The demand-driven design technique has served us well. One of the non-obvious advantages is that work can be interrupted at any stage and resumed at a later time without undue trouble, since at any moment the reasons for all decisions taken so far are obvious. A distinct disadvantage is that it reduces compiler design to a bookkeeper's job which lacks the fascination that attracts the majority of computer scientists. Perhaps the time has come to perform compiler design mechanically.

Starting from the design thus obtained, F. van Dijk wrote an ALEPH compiler in ALEPH, which was bootstrapped to ALICE. This compiler is available in ALEPH and in ALICE [VAN DIJK 82]. For those who have access to a Control Data Cyber, a processor from ALICE to COMPASS both in ALEPH and in COMPASS is available.

An independent ALEPH compiler was written by Csirmaz László of the

Mathematical Institute of the Hungarian Academy of Sciences, who also made a Hungarian translation of the ALEPH Manual [CSIRMAZ 77]. The compiler was bootstrapped onto the IBM 370 by Kósa Márton and Fuchs György.

1.3. On the structure of this thesis

The ALEPH project is moderately small as language projects go. Nevertheless the number of identifiable decisions taken in the design of the language and its compiler is very large. It would be out of the question to describe all these decisions with their arguments and interrelations. So some structure has to be discerned in the material to be able to present it.

It is tempting to say that the project has a tree structure: when we think about the compiler we do not think about the language design and when we think about an intermediate code we do not think about parsing techniques. In the higher regions of the tree this is satisfactory, but the nearer we get to the leaves, the more our view is obscured by interrelations and interferences: problems inside the language design cannot be described in isolation, those in the compiler even less. The tree turns into a directed graph.

It is, however, in these lower regions of the tree that the hard core of the design is to be found. Any description on a higher level remains fuzzy: observations on the design technique remain floating in the air unless supported by at least one example of that technique shown at work.

In an attempt to treat enough hard material in a sufficiently small space, two levels of description have been used. A first-level description of a node describes the sub-tree beneath that node, and, since this thesis is concerned with design techniques, it explains how the sub-tree was dealt with; it may identify new sub-trees, for which again a first-level description may be given in a later paragraph. Its purpose is to give the reader an impression of that part of the project. A second-level description explicitly describes the whole sub-tree involved and is concerned more with technical details than with a broad view. It serves to illustrate the design principles expounded in the first-level description of the same sub-tree.

A good example is the treatment of ALICE, the ALEPH intermediate code. The (first-level) description of the compiler (4.2) reviews some necessary concepts, one of which is ALICE. The chapter on ALICE (4.4) refers to the defining document, gives a short introduction to ALICE, identifies some problems and describes the technique used to solve them, all on the first level. The actual solving of the problems is then shown in detail in chapter 6.

1.4. Global view

The following survey of the contents of this thesis may be helpful.

The thesis.

> *A grammar can be interpreted as a program, which makes the grammatical formalism correspond to a programming language. ALEPH is a concretization of this idea. Detailed decisions are discussed and a well-structured machine-independent compiler is developed.*

4. On the design of the ALEPH compiler
   4.1. History of the compiler
   The original ALEPH compiler, which was derived from the CDL compiler producing ALGOL 60 on the EL-X8, was bootstrapped into producing COMPASS on the Cyber. It, in turn, helped bootstrapping the completely new machine-independent ALEPH compiler described in this thesis.
   4.2. The design technique
      4.2.1. Design criteria
      The issues were portability, minimal memory requirements and simplicity of design.
      4.2.2. The portability of the compiler
      The compiler produces ALICE, a special intermediate code tailored to ALEPH. The mapping from an ALEPH program to an ALICE program is completely machine-independent.
      4.2.3. The four stages of the design
      The task of designing the compiler was factorized into four subtasks, each of which was performed in bookkeeper's fashion.
      4.2.4. Evaluation
      Some parts of the design process were almost mechanical.
   4.3. The parser
   By using information streams on files wherever possible it keeps memory requirements low. It was derived interactively from an LL(1)-type grammar.
   4.4. On ALICE
      4.4.1. A short introduction to ALICE
      An ALICE program consists of a highly structured stream of macro calls, many of which are redundant on a given machine.
      4.4.2. The design of ALICE
      An attempt has been made to combine reasonable simplicity of machine-code generation with reasonable run-time efficiency of the code obtained. This resulted in some unusual data types, like the ALICE 'gate' (parameter transfer area).
      4.4.3. Problems with and modifications to ALICE
      The problems that cropped up when ALICE was used in practice are discussed and a technique to mend them, the 'parallel-script technique', is developed.
   4.5. Bootstrapping
   The practical application of ALICE in porting the compiler is explained in a linear notation.

5. The design of the ALEPH compiler
   The design of that part of the ALEPH compiler that produces the ALICE **value***s is given in full detail.*
6. Modifications to ALICE
   The development of the necessary modifications to ALICE is given in full detail.
7. References
8. Summary
9. Summary in Dutch
10. Curriculum vitae
11. Index

Appendix: ALEPH Manual
The first edition of the ALEPH Manual was written in 1973 [BOSCH, GRUNE & MEERTENS 73]; the version presented here is the fourth edition.

0. Preface
   The differences between the third edition and the present one are listed.
1. An informal introduction to ALEPH
   A small program for reading and evaluating integer expressions is derived in a tutorial manner from the grammar of the input. Most of the language facilities are touched upon.
2. Introduction to the Manual
   The syntactical description used is explained.
3. Program logic
   The language constructs that govern the flow of control are described in detail: rules, affixes, operations, affix-forms, terminators, compound-members, classifications and criteria for side-effects and failing.
4. Data declarations
   Concerns the language constructs that allow the declaration of global data: expressions, constants, variables, tables, stacks and files.
5. Externals
   The actual data handling in ALEPH is performed by 'externals', which do not belong to the language proper.
   5.1. User externals
      How to declare a (special-purpose) external not provided in the standard.
   5.2. Standard externals
      A number of actions are available without explicit declaration.
6. Pragmats
   Pragmats govern the behaviour of the compiler rather than that of the program.
7. The representation of programs
8. Examples

9. References
10. Index

1.5. Acknowledgements

It has been said that a thesis in computer science will cost fifteen man-years, and the present project is not far off that mark. Without the sustained effort of many people this thesis just would not have existed, and I realize with gratitude that Rob Bosch, Wim Böhm and Frank van Dijk have each given several years to the ALEPH project. Rob Bosch wrote the first (machine-dependent) ALEPH compiler, Wim Böhm designed ALICE, the *AL*eph *I*ntermediate *C*od*E*, and Frank van Dijk implemented the new ALEPH compiler.

Prof. A. van Wijngaarden, my promotor, has been an inspiring listener who has left me a great deal of much-valued freedom in organizing this text.

In 1970 Kees Koster started the CDL-project from which the ALEPH-project derives, and in 1982 he acted as coreferent for this thesis, thus spanning the complete project over more than a decade.

Hans van Vliet and Lambert Meertens gave the manuscript a careful reading and Sándor Nacsa showed interest in ALEPH at a moment when that commodity was in short supply.

I am grateful to the many friends who have given me mental support, especially to my wife Lily, who kept a steady faith in the eventual success of this venture.

The front cover was drawn by Tobias Baanders, in the spirit of ALEPH and recursion. In preparing the text of this thesis extensive use was made of the full power of UNIX [RITCHIE & THOMPSON 74], resulting in these phototypeset pages. The printing was carried out by D. Zwarst, J. Schipper, J. van der Werf, J. Suiker and F.J.C. Swenneker.

Gerard Kok has written a tutorial on ALEPH [KOK 77].

2. ON GRAMMARS

Some paragraphs in this thesis make extensive use of the concept of 'formal grammar' (or 'grammar' for short). We shall assume that the reader is more or less acquainted with formal grammars. An excellent exposition, both for the novice and for the expert, is given by J. Craig Cleaveland and R.C. Uzgalis [CLEAVELAND & UZGALIS 77]. For a thorough treatment of the subject the reader is referred to [HOPCROFT & ULLMAN 79]. A survey of the various notations in use in computer science is presented in [MARCOTTY & LEDGARD 76].

A grammar is a formal recipe for generating sentences (= sequences of symbols). The formal recipe consists of a number of formulas in a specific notation and of instructions (generally in informal English) on how to manipulate the formulas in order to generate the sentences. The exact form of the formulas depends on the type of the grammar, but a specific kind of formula, called "production rule", is always present. A production rule has a "name", often called its "left-hand-side" (LHS), and a "right-hand-side" (RHS). We separate the LHS and the RHS by a colon (':') and terminate the rule by a period ('.'). The RHS consists of one or more "alternatives", separated by semicolons (';'). An alternative consists of one or more "members", separated by commas (','). A member is either a name or epsilon (ϵ). If a member is a name, it may be the name of a production rule (the same or another one), or the name of a terminal symbol.

Another item that is always present is the "starting name", also called "initial symbol", "root", etc. We shall generally use the name **'text'** as the starting name.

In this thesis we shall meet mainly three types of grammars: context-free grammars, VW-grammars and affix grammars. Grammars and their constituents will be printed in **bold**.

2.1. The production mechanism

The purpose of a grammar is to describe (delineate) a set of sentences. It performs this service by being a recipe for producing all members of that set. Although the details of the production mechanism depend on the grammar type, the general process for generating a sentence is as follows.

We operate on a "sentential form", a sequence of members separated by commas. Our initial sentential form consists of the starting name. As long as the sentential form still contains a name of a production rule, we replace that name by one alternative from the RHS of that production rule. This process stops when the sentential form consists of names of terminal symbols and ϵs only. We cross out the ϵs, replace each name of a terminal symbol by its representation, and remove the separating commas.

The result of this process can be depicted as a tree: the root is the starting name, which branches into the members of its chosen RHS; each member branches again, etc. The leaves are the names of the terminal symbols. This tree is called the "parse tree" and it contains a record of the production process.

It should be noted that this process is not guaranteed to terminate for arbitrary choices of the alternatives. For some grammars the production process cannot terminate at all.

2.2. Top-down parsing

Often we have a sentence and we want to know whether it can be produced by a given grammar: the "recognition problem". Moreover, if it can, we generally want to know *how*, i.e., we want to reconstruct the parse tree: the "parsing problem". (Not all types of grammars allow these problems to be solved in general.)

The main two general ways of tackling the parsing problem are the 'bottom-up' and the 'top-down' methods.

In the bottom-up method we try to carry out the above procedure in the opposite direction: we search for RHSs we can recognize and then replace these by the corresponding LHSs. If we manage to reduce the sentence to the starting name, we have found a parsing. We shall make little use of this technique.

In the top-down method we try to imitate the production process which produced the sentence in the first place. We set out to generate all sentences and end immediately each attempt of which it has become clear that it will not lead to the desired goal. For a detailed description see, e.g., [AHO & ULLMAN 72, p. 285-301].

When we carry out this process deterministically, we try the alternatives of a given production rule in some order. One alternative A may seem very promising for a long time, thus leading us to continue the parsing attempt with further rules, try their alternatives, etc. At a certain moment the attempt may turn out to be a failure and then we have to find our way back so that we can try the successor, if any, of the alternative A ; this is called "backtracking".

The general top-down technique may be extremely expensive. There is, however, a simple way to cut the cost to a very acceptable level. We require the grammar to be such that at each production rule we can tell from the next k terminal symbols in the sentence which alternative to take. Consequently, we are never in doubt as to which alternative to try and we shall never have to backtrack. In particular there can be at most one parsing for the entire sentence: the grammar is unambiguous. A grammar that allows this simplification is 'of type LL(k)'. We shall often require a grammar to be of type LL(1).

The notion 'LL(k)' is treated extensively by D.E. Knuth [KNUTH 71]. For a short history of LL(k) grammars, see [AHO & ULLMAN 72, p.368].

2.3. VW-grammars

It is well known that every recursively enumerable language can be described through a general (type 0) phrase-structure grammar, but it is also true that if the language is not context-free, the grammars that describe it generally give little or no indication of the nature of that language. A good example is the language $L = \{a^n b^n c^n \mid n \geqslant 1\}$ for which the following phrase-structure grammar is cited [CLEAVELAND & UZGALIS 77, 1.3.4] (single-letter notion names have been replaced by more informative ones):

text: a symbol, b symbol, movable c;
a symbol, text, low b, movable c.
movable c, low b : marker, low b.
marker, low b: marker, movable c.
marker, movable c: low b, movable c.
b symbol, low b: b symbol, b symbol.
movable c: c symbol.

where **a-symbol** has the representation *a*, **b-symbol** has *b* and **c-symbol** has *c*.

A. van Wijngaarden has given another way to describe a recursively enumerable language, viz., through a two-level grammar [VAN WIJNGAARDEN 65]. To introduce the pertaining concepts and techniques we shall give here an informal construction of a VW-grammar for the above language $L = \{a^n b^n c^n \mid n \geqslant 1\}$.

We could describe the language L through a context-free grammar if grammars of infinite size were allowed:

text:a symbol, b symbol, c symbol;
a symbol, a symbol, b symbol, b symbol, c symbol, c symbol;
a symbol, a symbol, a symbol, b symbol, b symbol, b symbol,
c symbol, c symbol, c symbol;
... ...

We shall now try to master this infinity by constructing a grammar, which allows to produce the above grammar for as far as needed. We first introduce an infinity of names:

text:ai, bi, ci;
aii, bii, cii;
aiii, biii, ciii;
... ...

with three infinite groups of rules:

ai: a symbol.	**bi: b symbol.**	**ci: c symbol.**
aii: a symbol, ai.	**bii: b symbol, bi.**	**cii: c symbol, ci.**
aiii: a symbol, aii.	**biii: b symbol, bii.**	**ciii: c symbol, cii.**
... ...	... ...	... ...

Next we introduce a special kind of name called "metanotion". Rather than being capable of producing (part of) a sentence in the language, it is capable of producing (part of) a name in a grammar rule. In our example we want to catch the repetitions of **i**s in a metanotion **N**, for which we give a context-free production rule (a "metarule"):

N :: i ; i N .

Note that we use a slightly different notation for metarules: LHS and RHS are separated by a double colon (::) and members are separated by a blank ( ).

Now the four infinite groups of rules collapse into four *finite* rule templates called "hyper-rules".

text: a N, b N, c N.

a i: a symbol.	**b i: b symbol.**	**c i: c symbol.**
a i N: a symbol, a N.	**b i N: b symbol, b N.**	**c i N: c symbol, c N.**

Each original rule can be obtained from one of the hyper-rules by substituting a production of **N** for each occurrence of **N** in that hyper-rule, *provided that* the same production of **N** is used consistently throughout. To distinguish them from normal names these half-finished combinations of small letters and metanotions (like '**a N**' or '**b i N**') are called "hypernotions".

We can also use this technique to condense the finite parts of a grammar:

N :: i ; i N .
A :: a ; b ; c .

text: a N, b N, c N.
A i: A symbol.
A i N: A symbol, A N.

Again the rules of the game require that the metanotion **A** be replaced consistently.

This grammar gives a clear indication of the language it describes: once the 'value' of the metanotion **N** is chosen, production is straightforward.

It is important to note that although this tutorial derivation uses infinities, the final grammar is finite and so is the production process: for the production of a particular element of L only a finite number of production rules need to be generated.

The metanotion mechanism is so suitable for carrying context information that all the context conditions (identification, data-type consistency, etc.) of a programming language can be described by it. The context conditions are often enforced by blocking production paths which would lead to sentences that violate these conditions. On such a path a name occurs for which no production rule can be generated from any template: we are in a "blind alley". Other mechanisms are the "infinite production path", in which an attempt to violate a context condition prevents termination of the production process, and the "repeated metanotion", in which the repetition of a metanotion forces a match in a sentential form.

VW-grammars incorporating all context conditions exist for ALGOL 68 [VAN WIJNGAARDEN 75] and for ALEPH [GLANDORF, GRUNE & VERHAGEN 78]. The techniques used are explained in detail by J. Craig Cleaveland and R. C. Uzgalis in [CLEAVELAND & UZGALIS 77]. M. Sintzoff has proved that there exists a VW-grammar for every recursively enumerable language [SINTZOFF 67].

The use of a VW-grammar can be extended to include the description of the semantics of the generated language [CLEAVELAND & UZGALIS 77, 4.5] or to produce results directly without the intervention of a programming language [VAN WIJNGAARDEN 81].

2.4. Affix grammars

The parsing problem for VW-grammars cannot be solved in general [SINTZOFF 67, Corrolary 2]. If we try to derive a parser from a VW-grammar by techniques analogous to those used in 2.2, we run into problems. Normally a LHS corresponds to the name of a parsing procedure in the parser, but the LHS of

A i N: A symbol, A N.

is not a procedure name but a template to generate an infinity of names. Often replacing a metanotion by a parameter helps, but even that fails in this case.

This situation can be remedied by using an 'affix grammar', a different type of two-level grammar, formulated by C.H.A. Koster. Although the parsing problem for general affix grammars cannot be solved either, manageable restrictions can be formulated on them to yield a subset, the "well-formed" affix grammars, for which the parsing problem *can* be solved. The properties of affix grammars are described in a harsh and forbidding formal form in [KOSTER 71b]. A more palatable treatment of a slightly modified form is given by H. Meijer [MEIJER 80] (or see [WATT 77]).

Affix grammars have the same expressive power as VW-grammars; P. Kühling has shown that the semantics of a programming language can be suitably expressed by means of an affix grammar [KÜHLING 78]. They differ from VW-grammars mainly in two points: there is a strict separation between the name of a production rule (its "handle") and the metanotions it carries (its "affixes"), and there is a strict separation between rules that produce (part of) the sentence and rules that enforce context conditions by checking affixes (the "primitive predicates").

A primitive predicate, which has affixes like a normal rule, contains a total recursive function, which will produce ϵ when the affixes satisfy the context condition implemented by this primitive predicate, and otherwise the forbidden symbol ω. The set $L = \{a^n b^n c^n \mid n \geqslant 1\}$ is then produced by the grammar in Fig. 1.

A number of conditions are imposed on an affix grammar to make it "well-formed". These conditions effect a division of the affixes in those with known values (technically called "inherited affixes") and those with undecided values ("derived affixes"); moreover, for each primitive predicate an effective procedure is required, which, given its inherited affixes, will generate a choice for its derived affixes. Thus a structure is created for which a parser can be derived, as proved in [KOSTER 71b, 8].

A related notion is that of 'attribute grammars' [KNUTH 68].

Since any program can be considered as a suitably coupled combination of a context-sensitive sentence parser and a context-sensitive sentence generator, the idea suggests itself to write programs in a form analogous to affix grammars. The ALEPH project is an attempt to make this idea practical. The train of thought that has led from VW-grammars to ALEPH is given in 3.3.

N: 1; N 1.
M: 1; M 1.
A: a; b; c.
B: a; b; c.

text + N: $ a production rule
    list + N + a, list + N + b, list + N + c.

list + N + A: $ a production rule
    where is zero + N;
    letter + A, where is decreased + M + N,
        list + M + A.

letter + A: $ a production rule
    where is + A + a, a symbol;
    where is + A + b, b symbol;
    where is + A + c, c symbol.

where is zero + N: $ a primitive predicate
    λx: $(x = 0 \rightarrow \epsilon,\ x \neq 0 \rightarrow \omega)$.

where is decreased + N + M: $ a primitive predicate
    $\lambda x\ \lambda y$: $(x = y - 1 \rightarrow \epsilon,\ x \neq y - 1 \rightarrow \omega)$.

where is + A + B: $ a primitive predicate
    $\lambda x\ \lambda y$: $(x = y \rightarrow \epsilon,\ x \neq y \rightarrow \omega)$.

Fig. 1.

3. ON THE DESIGN OF THE ALEPH LANGUAGE

3.1. History of affix grammars

Affixes were first used in 1962 by L. Meertens in writing a context-free grammar for part of the English language. Such a grammar tends to be very repetitive and affixes were found a welcome means of abbreviation. The meta-grammars of the affixes were finite-choice and the resulting grammar was indeed context-free.

L. Meertens and C.H.A. Koster converted this affix grammar by hand into a sentence-producing program, which ran on the EL-X1 of the Mathematical Centre. It produced sentences like 'I had been showing the extraordinary long tooth that I who had brightened always must have wanted'. Soon a simple Dutch version followed, by Koster. It produced the hilarious but untranslatable 'kikvorsen zijn grote kikkers'.

Around 1966 Meertens wrote an affix grammar for composing music, in which the affixes were integers on which arithmetic was done in special rules. This grammar was no longer context-free: affixes had passed from an abbreviation mechanism to a control mechanism.

Meanwhile Koster worked on the parsing and translating of natural languages by means of affix grammars. In [KOSTER 65] a translator from (partial) English to German is described which can cope with sentences like: 'the woman in whose house i live has a small beautiful garden too', which resulted in the stilted German phrase 'die frau in deren hause ich wohne hat auch einen kleinen schoenen garten'.

In the years that followed Koster applied the experience with affix grammars, gained in these experiments, to ALGOL 68, which was described by a VW-grammar (3.3.1.1). The desire to generate the compiler (or at least the parser) automatically, resulted in the development of CDL (*C*ompiler *D*escription *L*anguage) [KOSTER 71a]. For the use of CDL to describe parts of a compiler see [KOSTER 72]. In 1971 a formal definition of affix grammars appeared in [KOSTER 71b], in all its technical detail.

In the beginning of 1972 Koster left the Mathematical Centre. D. Grune, R. Bosch and L. Meertens took over the project and turned the compiler-description language into a programming language: ALEPH [GRUNE, BOSCH & MEERTENS 74]. Koster continued the development of CDL and its successor CDL2 in Berlin [DEHOTTAY et al. 76].

Both CDL and ALEPH are based on top-down parsers. D. Crowe published a bottom-up parser for affix grammars in 1972 [CROWE 72], which was improved by A.P.W. Böhm in 1974 [BÖHM 74]. D.A. Watt has given a technique to extend any given parser-generating method for context-free grammars into a parser-generating method for affix grammars [WATT 77].

3.2. The design philosophy

3.2.1. Some thoughts on producing correct programs

When a human speaker (or writer) conveys a message to a human listener (or reader), the receiver immediately subjects the message to a reasonability check, based on his extensive knowledge of the world. When, for instance, a newspaper reader finds New York called 'the capital of the US', he will think that somebody made a mistake, not that he missed a major constitutional development. This error tolerance of the listener is very useful in that it allows the speaker to express complicated things in a

few words in a sloppy way. If I ask at the pastry shop: 'Can I have another peach pie, just like the one I had yesterday', I generally get results, even if it was an apricot pie and the shop was closed yesterday.

Our entire way of communication is based on the fact that we are communicating with a reasonable partner whose knowledge of the world is comparable to ours. We expect that if we happen to say something (formally) nonsensical, we will either be understood anyhow or somebody will ask back what we meant. Our messages are never more than 'almost correct'. We see that our experience with daily communication rests, among other things, on two phenomena: plausibility check ('They can't mean that!') and feed-back ('Can you be here tomorrow at eight?' 'You mean AM or PM?').

In the communication with a computer these two phenomena are largely absent, and consequently we cannot expect our daily communication techniques to work properly for communicating with a computer: a computer will not work on a handful of 'almost correct' instructions. On the contrary, we expect a good man-machine communication technique (a programming language) to deviate considerably from a natural language, and if it happens to fit in well with everyday thinking (i.e., accomodates sloppiness well), we do not consider that an asset. As we have seen, a natural language is a means of producing efficiently 'almost correct' messages, sufficiently correct for practical use; a programming language, however, should supply methods for producing 'completely correct' messages and we should be willing to pay for the loss of efficiency in the message production (cf. also [HILL 72]).

3.2.1.1. The methods

A good programming language should supply the user with methods that can be handled with reasonable mental effort and that, with reasonable ease, lead to completely correct formulations. ALEPH is based on three such methods, well-known from literature and practice:

1 the selection of an applicable alternative out of a list of them, through the fulfilment of an entry criterion,

2 the decomposition of a problem into a sequence of sub-problems, any of which may be similar to the original problem,

3 the packaging of a list of alternatives into a named procedure.

The first method is similar to the 'guarded commands' [DIJKSTRA 75], although details of the semantics differ. The second is widely known under names like 'hierarchical programming', 'top-down approach', 'divide & conquer', etc. The third is the traditional procedural abstraction mechanism.

It is important to note that all three mechanisms can be found in the structure of a context-free grammar, where a rule (i.e., a procedure) consists of a list of alternatives, each of which is decomposed into the names of other rules. This analogy, which is a cornerstone in the design of ALEPH, is elaborated upon in 3.3.

3.2.1.2. The use of redundancy

One way to increase the reliability of communicated messages is to supply them with redundancy. The function of this redundancy is to dilute the universe of possible messages to the effect that if a message is damaged in the communication there is a high probability that it turns into a non-message, detectable by the receiver. A simple

way to achieve this is to send the message twice in a different coding, e.g., once in Dutch and once in Hungarian.

This does not seem to have any bearing on programming languages, since there is no noisy channel between the sender of the message (the programmer) and the receiver. The noisy channel, however, is somewhere else, between the intention of the programmer and the formalization of this intention as a program. Again we benefit if the intention is transmitted more than once, since this allows the receiver (the computer) to do consistency checking. The obvious example is the specification of the data types of entities in the program, in particular of the formal parameters of a procedure. When a call of a procedure is met, the types of its parameters are known from two different sources and a consistency check can be made. See also [FEUERHAHN & KOSTER 78, 2.1].

In addition to data-type checking ALEPH has rule-type checking, based on information about side-effects and/or the possibility of failure, known along different paths (3.3.2.5).

3.2.2. Machine-independence and portability

A major issue in the design was the portability of ALEPH programs, including the compiler. The problem has been approached by the use of a machine-independent intermediate code specific to ALEPH, named "ALICE". Detailed issues in portability are discussed in 3.4; a short survey of ALICE is given in 4.4.

This approach is in sharp contrast to the technique through which CDL and CDL2 achieve portability, viz., open-endedness [STAHL 78]. All data manipulation in a CDL program is done through calls to rules declared in that program. The programmer has the choice of either declaring a rule in terms of CDL-constructs or declaring it as a "macro", in which case he has to supply a macro-body with code specific to the target-language of his machine. Portability is then achieved by rewriting the macro-bodies of the program (and those of the CDL compiler).

ALEPH, on the other hand, has built-in data-handling primitives (like **stack-declaration**s, **extension**s, standard-externals, etc.) and the programmer is expected to express his algorithms entirely in these terms. These primitives are supported by ALICE and portability is now achieved by implementing ALICE on the new machine, after which both the ALEPH compiler and the user program will run (4.5.2).

If the data-handling the user requires cannot be reasonably expressed in the predefined primitives (e.g., reaching specific system facilities), the user can escape to a macro level through an **external-rule-definition**, but the portability of the resulting program is then jeopardized.

3.3. From VW-grammar to ALEPH*

ALEPH has the interesting quality that it is large enough not to be dismissed as a toy language and small enough to keep the task of designing it intellectually manageable (although barely so).

Therefore an account of the design of ALEPH is interesting not only because of its results, a language with a very simple but powerful flow-of-control, in which the uninitialized-variable problem is solved and in which side effects are under full

*This section is a revision of [GRUNE 81].

control, but also because the way in which these results are obtained is open to examination.

In this chapter we shall give an exposition of the designing of ALEPH. We shall not completely follow the historical development, since that included many side tracks without issue (e.g., a satisfactory parameter-passing mechanism was found only after much experimentation). A survey of the line of argument is given in the directed graph in Fig. 2. The bubbles contain concepts; the arrows can be read as 'leads to' or 'is a prerequisite for'. The triangles, which have no predecessors, contain ideas that come from the outside world; the parallelograms, which have no successors, contain (hopefully desirable) results for that outside world.

Figure 2 is a simplification of reality: more arrows could be drawn, but the main ones are included. The picture bears resemblance to the dependency graph of modules in a large program; several layers can be distinguished: programming language, flow-of-control, affixes, affix rules, globals.

Inside these levels the dependency of the concepts is fairly badly structured, as can be expected of an object that was not designed according to firm design rules.

Little is known about design rules for programming languages. In essence design rules serve to reduce the intellectual complexity of a task. Traditional means are: imposing a structure, divide-and-conquer, defining interfaces, etc. Hardly any of these applies to the design of programming languages. The most successful principle is still orthogonality, which also has its problems. It does not allow the designer to distinguish between the cheap and the expensive, and its consistent application is difficult.

Our discussion will lead us from VW-grammars through affix grammars to ALEPH and conventional programming languages. Each of these fields has its own (traditional) terminology and often a concept in one field will reappear in the next (in a slightly modified form) under a different name. It may be helpful for the reader to refer to Fig. 3 for the approximate relations.

3.3.1. Turning VW-grammars into a programming language

3.3.1.1. Two-colour grammars

A VW-grammar is a special type of phrase-structure grammar, which retains some of the important properties of a context-free (CF) grammar. We can use a CF grammar to describe *any* language, provided that this grammar may have infinitely many production rules; every actual production of a desired sentence in the language, however, needs only a finite number of them. In essence a VW-grammar is a recipe for generating such an infinity of CF production rules. In deriving a sentence we keep the derivation finite by generating only those rules that we actually need for the production of that sentence.

A VW-grammar has the following main constituents:

- the metarules, a collection of (interrelated) CF grammars, each producing a language for a specific metanotion;
- the hyper-rules, a collection of templates from which to form (an infinity of) CF production rules.

A CF production rule is derived from a hyper-rule by replacing consistently each of the metanotions it contains by a terminal production of that metanotion.

VW-grammars:	affix grammars:	ALEPH:	conventional programming languages:
grammar	grammar	program	program
some initial hypernotion	initial symbol	root	main procedure
hyper-rule	rule	(global) rule	procedure
invisible production	primitive predicate	external rule	built-in function
	left-hand-side, LHS	rule head	procedure heading
	right-hand-side, RHS	rule body	procedure body
	alternative	alternative	control structure
may produce empty	may produce ϵ	always succeeds	always yields *true*
is a blind alley	produces ω	fails	yields *false*
hypernotion	affix expression	affix form, rule call	call
metarule	affix rule	— —	— —
metanotion	affix, bound affix, free affix	affix, formal affix, local affix	parameter, formal parameter, local parameter
symbol	terminal symbol	input/output operation	input/output operation

Fig. 3.

Let us now introduce the notion of a 'two-colour' VW-grammar. We start from a VW-grammar R, which produces sequences of symbols in red. We then take a second VW-grammar P, which shares part or all of its metarules with R and which produces its symbols in blue (or in a different alphabet if you wish). We now combine the two grammars and insert hypernotions of P in hyperalternatives of rules of R: the resulting grammar produces sentences in mixed red and blue text.

If it now so happens that a hypernotion of P shares one or more metanotions with some of its neighbours that belonged to R, then the production of blue text is controlled by the same choice of metanotion substitutions as that of the red text, and the red and blue pieces of text will become correlated.

Figure 4 shows a two-colour grammar for the language $\{red\text{-}a^n\ blue\text{-}b^n\ blue\text{-}c^n \mid n \geqslant 0\}$; this language cannot be produced by a CF grammar and the distribution of information through metanotions is essential. We shall gradually transform this example grammar until it has become an ALEPH program that recognizes the red text and produces the blue one. To smooth the transitions in the explanation the starting point is more complicated than strictly necessary: context conditions are stored in 'invisible productions'. A VW-grammar for the above language is given as grammar Q in [CLEAVELAND & UZGALIS 77, 3.4]; invisible productions are explained in [CLEAVELAND & UZGALIS 77, 3.5].

TCG 1:

```
N :: N n; .
ABC :: a; b; c.

text: red N a, blue N b, blue N c.

red N ABC:
    red symbol ABC, red N1 ABC,
        where rd N1 plus one is N;
    where rd N is zero.
red symbol ABC: red letter ABC symbol.

where rd N plus one is N n: where true.
where rd   is zero: where true.

blue N ABC:
    where bl N is zero;
    blue symbol ABC, where bl N1 is N minus one,
        blue N1 ABC.
blue symbol ABC: blue letter ABC symbol.

where bl N is N n minus one: where true.
where bl   is zero: where true.

where true: .
```

Fig. 4.

A possible production of *TCG* 1 is (with **N** = **nnn** in **text**):

red-a red-a red-a blue-b blue-b blue-b blue-c blue-c blue-c

3.3.1.2. A top-down parser

It is well known that a CF grammar can be turned mechanically into a recognizer for the language it produces (e.g., [KNUTH 71]). In the general case this can be inefficient, but if sufficient restrictions are put on the CF grammar, neat recognizers result. Specifically, the LL(1) restriction leads to an efficient top-down parser, which, as a program, has virtually the same form as the original grammar.

This suggests that it may be possible to consider the red part of the two-colour grammar *TCG* 1 (which, in a sense, *is* LL(1)) as a top-down parser for the red text, while at the same time retaining the producing nature of the blue part. If we do this, we are led to consider the occurrences of metanotions in hypernotions as parameters. We shall not worry at the moment about the exact parameter-passing mechanism; for the time being it can be thought of as 'call-by-name'. This brings us to the grammar/program of Figure 5.

P 1:

text: read N a, print N b, print N c.

read N ABC:
    read symbol ABC, read N1 ABC,
        where rd N1 plus one is N;
    where rd N is zero.
read symbol ABC: absorb letter ABC.

where rd N1 plus one is N: set N to N1 plus one.
where rd N is zero: set N to zero.

print N ABC:
    where pt N is zero;
    print symbol ABC, where pt N1 is N minus one,
        print N1 ABC.
print symbol ABC: produce letter ABC.

where pt N1 is N minus one: set N1 to N minus one.
where pt N is zero: is N zero.

Fig. 5.

When we read it as a VW-grammar we encounter two new production rules, which can easily be defined:

produce letter ABC: blue letter ABC symbol.
absorb letter ABC: red letter ABC symbol.

The grammar *P* 1 then produces the same language as grammar *TCG* 1.

However, when we read it with the firm conviction that it is a program, meaning begins to attach itself to various constructs. To perform **text**, read *N a*s, then print *N b*s, then print *N c*s. To read *N ABC*s, we have the choice between two alternatives, which we shall try in order. We attempt to read a symbol *ABC*, and if we succeed we read *N* 1 *ABC*s and set *N* to *N* 1 plus one; otherwise (if we cannot read a

symbol *ABC*) we set N to zero. In this same vein we can understand the rest of the program, which prints N *b*s and N *c*s.

Here we interpret the production rules of the grammar as production rules of the program, which either succeed or fail. A special interpretation is necessary for *produce letter* and *absorb letter*:

```
produce letter ABC:
    $ a procedure that appends the letter ABC to the output.
    .

absorb letter ABC:
    $ a procedure that examines the first character of the
    $ input:
    $    if that character is the letter ABC, it removes the
    $    first character from the input and succeeds;
    $    otherwise, it fails.
    .
```

At this point the reader will have gathered that we have cheated. The above example was rigged so that its interpretation as a program suggested itself. If we take a different VW-grammar, e.g., the one describing ALGOL 68 [VAN WIJNGAARDEN 75], the above line of thought fails miserably, on several points. Among the reasons for this are:

- Hypernotions cannot in general be identified by some characteristic part. (The ALGOL 68-grammar is an exception: it has very few points where one is in doubt).
- Confusion arises as to where the terminal production of a metanotion begins or ends inside a hypernotion.
- Values of metanotions are used before they are known.

There is, however, a type of two-level grammar related to VW-grammars for which the parsing problem can be solved: the affix grammars.

3.3.1.3. Affix grammars

Affix grammars are defined by C.H.A. Koster [KOSTER 71b]; this definition is slightly modified and explained well in [WATT 77]. Koster shows, given an affix grammar that is 'well-formed' (see below), how to construct a parser for the language it generates. Most constituents of a VW-grammar also exist in an affix grammar. For a list of correspondences see 3.3. The principal differences between affix grammars and VW-grammars are:

- a hypernotion consists of a characteristic name, its 'handle', followed by one or more metanotions, called 'affixes', and
- context conditions are enforced by special rules called 'primitive predicates', which can be thought of as affix checkers.

A 'primitive predicate' is similar to a (normal) rule in that it has affixes; but rather than producing its output by specifying affix forms and terminal symbols, it contains a total recursive function T, the "associated function", which, depending on the affixes, will produce either 'empty' (ϵ) or the forbidden symbol (ω).

Affixes occurring in the LHS of a rule are called 'bound' affixes to that rule; affixes that occur in the alternative(s) in the RHS only are called 'free'.

The well-formedness criterion requires (among other things) that all occurrences of affixes can be divided into two groups, the 'derived' (δ) and the 'inherited' (ι) affixes, under the following conditions:

- if a bound affix B of a rule is inherited, all occurrences of B in the RHS of that rule are inherited;
- if a bound affix B of a rule is derived, then the textually first occurrence of B in each alternative in the RHS of that rule is derived and all others are inherited;
- the textually first occurrence of a free affix F in each alternative in the RHS of a rule is derived and all others are inherited;
- for each primitive predicate with derived affixes D, inherited affixes I and associated function T, a total recursive function is given which will calculate D from I such that $T(I,D)$ succeeds (i.e., produces ϵ).

The first three requirements ensure that affixes can be interpreted as input- and output-parameters in a proper way; the last requirement makes it possible to reconstruct during parsing the context that was enforced during production.

An affix grammar equivalent to $TCG\,1$ is shown in Figure 6a/b. To satisfy the well-formedness requirement this text must be augmented by a list of functions, one for each primitive predicate, which calculate the derived affixes from the inherited ones. They are (in the form <name, domain of the inherited affixes, domain of the derived affixes, function>):

<where rd plus one is, (N), (N), $\lambda x\colon x + 1$**>,**
<where rd is zero, (), (N), 0**>,**
<where is, (ABC, ABC1), (), $\lambda x\,\lambda y\colon (x = y \rightarrow \epsilon,\ x \neq y \rightarrow \omega)$**>,**
<where bl is minus one, (N), (N), $\lambda x\colon x - 1$**>,**
<where bl is zero, (N), (), $\lambda x\colon (x = 0 \rightarrow \epsilon,\ x \neq 0 \rightarrow \omega)$**>**

They correspond to the '*set N to ...* ' in $P\,1$.

A more convenient variant of the affix grammars are the 'extended affix grammars' [KÜHLING 78], originally defined by D.A. Watt, in which the primitive predicates have been abandoned, and in which affix positions can be occupied by paranotions rather than by metanotions. Again, there are well-formedness conditions if the grammar is to be used in syntax analysis. Since extended affix grammars have played no role in the design of ALEPH, they will not be treated here any further.

3.3.1.4. CDL

It is simple to convert the affix grammar $AG\,1$ into a program; it will nevertheless be clear to the reader that affix grammars as such are less than attractive as a programming language. There are, however, some bright points: many of the least appetizing parts of the text exist only for the benefit of the description mechanism in [KOSTER 71b], and the similarity between part P of $AG\,1$ and the tentative program $P\,1$ is striking; moreover, parts of the text are redundant:

- V_n can be derived from P.
- Q and V_t can be derived from S and P.
- A_n and A_t follow from R.

AG 1:

\$ V_n: the non-terminal symbols
text, red, red symbol, blue, blue symbol.

\$ V_t: the terminal symbols
red-a, red-b, red-c, blue-a, blue-b, blue-c.

\$ A_n: the non-terminal affix symbols
N, N1, ABC, ABC1.

\$ A_t: the terminal affix symbols
n, a, b, c.

\$ Q: the primitive predicate symbols
where rd plus one is, where rd is zero, where is,
where bl is minus one, where bl is zero.

\$ E: the initial symbol
text.

\$ R: the affix rules
N:: N n; .
N1:: N.
ABC:: a; b; c.
ABC1:: ABC.

\$ S: the 'control' set; each quintuple contains:
\$ the name of a non-terminal or primitive predicate symbol,
\$ the number of affixes,
\$ the types of the affixes (derived or inherited),
\$ the domain of the affixes, and
\$ the associated function

<**text, 0, (), (), $\varnothing$**>,
<**red, 2, (δ, ι), (N, ABC), $\varnothing$**>,
<**red symbol, 1, (ι), (ABC), $\varnothing$**>,
<**where rd plus one is, 2, (ι, δ), (N, N1),**
$\lambda x\, \lambda y$: $(x + 1 = y \rightarrow \epsilon,\ x + 1 \neq y \rightarrow \omega)$>,
<**where rd is zero, 1, (δ), (N),**
λx: $(x = 0 \rightarrow \epsilon,\ x \neq 0 \rightarrow \omega)$>,
<**where is, 2, (ABC, ABC1), (ι, ι),**
$\lambda x\, \lambda y$: $(x = y \rightarrow \epsilon,\ x \neq y \rightarrow \omega)$>,
<**blue, 2, (ι, ι), (N, ABC), $\varnothing$**>,
<**blue symbol, 1, (ι), (ABC), $\varnothing$**>,
<**where bl is minus one, 2, (δ, ι), (N, N1),**
$\lambda x\, \lambda y$: $(x = y - 1 \rightarrow \epsilon,\ x \neq y - 1 \rightarrow \omega)$>,
<**where bl is zero, 1, (ι), (N),**
λx: $(x = 0 \rightarrow \epsilon,\ x \neq 0 \rightarrow \omega)$>.

Fig. 6a.

```
$ P: the rules
    text: red + N + a, blue + N + b, blue + N + c.

    red + N + ABC:
        red symbol + ABC, red + N1 + ABC,
            where rd plus one is + N1 + N;
        where rd is zero + N.
    red symbol + ABC:
        where is + ABC + a, red-a;
        where is + ABC + b, red-b;
        where is + ABC + c, red-c.

    blue + N + ABC:
        where bl is zero + N;
        blue symbol + ABC, where bl is minus one + N1 + N,
            blue + N1 + ABC.
    blue symbol + ABC:
        where is + ABC + a, blue-a;
        where is + ABC + b, blue-b;
        where is + ABC + c, blue-c.
```

Fig. 6b.

- When affix passing is implemented as call-by-name, the information about derived and inherited becomes immaterial (except for checking purposes); consequently all entries in S that concern members of V_n can be deleted.
- We can get rid of the metarules R by observing that the languages produced by the members of R are CF, and by making the sweeping statement that any language can be mapped on the integers: only integer values are necessary as affixes. (If we try this in practice we soon run into integer overflow, so eventually other means have to be devised.)

This reduction leaves us with P, E and the primitive-predicate descriptions in S. The latter can be implemented as macros, allowing escapes to a different regime, the total recursive functions with output parameters. A notation could be:

```
P2:
    INITSYM text.

    MACRO where rd plus one is = " '2' := '1'+1 ",
        where rd is zero = " '1' := 0 ",      $ a derived affix
        where is = " '1' = '2' ",
        where bl is minus one = " '1' := '2'-1 ",
        where bl is zero = " '1' = 0 ".       $ an inherited affix

                                              $ P, same as P of AG1 (Fig. 6b)
```

A few more steps along these lines will lead us to CDL [KOSTER 71a] and to its successor CDL2 [DEHOTTAY et al. 76].

In the remainder of this chapter we shall follow the line of thought that has led to ALEPH.

3.3.2. From affix grammar to ALEPH

Like in CDL we shall restrict ourselves to top-down (recursive descent) parsers, since they lead more easily to programming languages than bottom-up parsers. Bottom-up parsers for affix grammars have been constructed by D. Crowe [CROWE 72] and A.P.W. Böhm [BÖHM 74].

We shall now investigate the consequences of interpreting a grammar as a program. Although the affix grammar $AG\,1$ can easily be converted into a program, it will be clear that affix grammars are still a far cry from a usable programming language. We have 'primitive predicates', which form a kind of language inside the language. The global flow-of-control may be obvious but details about the local flow-of-control (i.e., inside a rule) have to be decided. The exact nature of affixes is open to negotiation. The affix rules describe data structures, but their form will depend on decisions about the affixes.

These issues are treated in the following paragraphs.

3.3.2.1. Global flow-of-control

The global flow-of-control relies completely on rules calling rules (recursively); since there is only one level of rules and rules cannot occur as parameters (nor be assigned to 'rule variables'), the program is a directed graph; the starting point is the ROOT. This has the great advantage that many properties of the program can be derived mechanically (e.g., recursion, global side effects). Together with the fact that affixes cannot be expressions that call user-defined rules, it also obviates the need for a display-like mechanism for affix-passing.

On the other hand it means that the rule-calling and affix-passing mechanism will be used heavily and that efficiency will be an important factor in the design of both.

3.3.2.2. Finding a place for the primitive predicates

The first three reductions mentioned above (3.3.1.4) are harmless. We shall postpone the decision about the affix-passing mechanism to 3.3.3.1 and incorporate the ι/δ information in the rule heads in P; an ι-affix (input affix) is marked by a *pre*fixed $>$, a δ-affix (output affix) by a *post*fixed $>$.

Next we realize that the number of primitive predicates can often be greatly reduced by describing their effect (producing ϵ or ω) in hyper-rules. For instance, the effect of

$$<\textbf{where prime, 1, } (\iota)\textbf{, (N), } \lambda x\textit{: x is prime} \rightarrow \epsilon\textit{, x is non-prime} \rightarrow \omega>$$

can be expressed in hyper-rules as follows (integral constants are used instead of sequences of **n**s):

where prime + N:
    where no divisor at or over + N + 2.

where no divisor at or over + N + N1:
where is + N + N1;
where indivisible + N + N1,
where plus one is + N1 + N2,
where no divisor at or over + N + N2.

where is + N + N1: ...
...

Many full-size examples of this technique can be found in [VAN WIJNGAARDEN 75, ch. 7] and in [GLANDORF, GRUNE & VERHAGEN 78]. This suggests the possibility of using a fixed set of metarules for every grammar, i.e., to supply a fixed set of data types in the programming language (theoretically this is no restriction, since it has been demonstrated that every VW-grammar can be rewritten so that only a fixed set of metarules remain [VAN WIJNGAARDEN 74]). These data types are then supported by a predefined set of predicates on them, the 'externals'. The choice of this set is treated in 3.5.

The RHS of a rule may contain both affix forms and terminal symbols; we shall simplify this situation by introducing two rules, *absorb* and *produce*. The affix form *absorb* + *ABC* looks at the next character in the input stream; if it is equal to *ABC*, *absorb* + *ABC* absorbs it and succeeds; otherwise it fails and leaves the input stream unaffected. The affix form *produce* + *ABC* produces the character *ABC*. Together they replace the absorption and production mechanism implied in the functioning of a two-colour grammar.

We shall change the keyword *INITSYM* to *ROOT*; the end of the text will be marked with an *END*. Our program is shown in Figure 7 (character constants are quoted with slashes /). Note that characteristic strings have been supplied in the *EXTERNAL* declarations, which enable the identification of the proper routines outside the program.

3.3.2.3. Local flow-of-control

Local flow-of-control is the flow-of-control inside a rule once it is called due to global flow-of-control rules. Since global flow-of-control is trivial (3.3.2.1), we shall use simply 'flow-of-control' for 'local flow-of-control'.

The parsing problem for affix grammars can be solved by a general top-down parser [KOSTER 71b, 8]. The flow-of-control rules in such a parser are:

General parser rules:
- Call the initial rule; iff it succeeds, the input belongs to the language.
- A rule is 'called' by trying the alternatives in its RHS for applicability and calling each applicable alternative.
- An alternative is always 'applicable' (see note below).
- An alternative is 'called' by calling its rules in textual order as long as these rule calls succeed.
- An alternative 'succeeds' iff all of its rule calls succeed.

P 3:

```
ROOT text.

EXTERNAL set to plus one + >N + N1> = "INCR",
    set + >N + N1> = "SET",
    set to minus one + >N + N1> = "DECR",
    equal + >N + >N1 = "EQUAL".

text: read + N + /a/, print + N + /b/, print + N + /c/.

read + N> + >ABC:
    read symbol + ABC, read + N1 + ABC,
        where rd plus one is + N1 + N;
    where rd is zero + N.
read symbol + >ABC: absorb + ABC.
where rd plus one is + >N + N1>: set to plus one + N + N1.
where rd is zero + N>: set + 0 + N.

print + >N + >ABC:
    where pt is zero + N;
    print symbol + ABC, where pt is minus one + N1 + N,
        print + N1 + ABC.
print symbol + >ABC: produce + ABC.
where pt is minus one + N> + >N1: set to minus one + N1 + N.
where pt is zero + >N: equal + N + 0.

END
```

Fig. 7.

- A call to a production rule *R* 'succeeds' iff *R* has at least one applicable alternative that succeeds.
- A call to a primitive predicate *P* may succeed or fail depending on the result of the evaluation of the total function of *P*.

Note: these rules are more complicated than necessary, since the notion of applicability is superfluous; we shall, however, need this notion in our further discussion.

The implementation of the above flow-of-control rules requires automatic backtracking (3.3.1.2). A traditional way to avoid backtracking is to require the grammar to be of type LL(1). So we have two options:

- either supply a backtracking facility;
- or refuse backtracking and require the affix grammar to be of type LL(1).

ALEPH is intended for the writing of production software; here any backtrack problems should be solved once at the writing desk, rather than over and over again when the program is run. This has led us to choose the second option.

Now what does it mean for an affix grammar itself to be of type LL(1)? It should be borne in mind that the LL(1)-property is important only because it allows simple

flow-of-control rules for a backtrack-free deterministic parser. We shall therefore take these rules as a starting point:

- Call the initial rule; iff it succeeds, the input belongs to the language.
- A rule is 'called' by trying the alternatives in its RHS for applicability and calling an applicable alternative (there can only be one such alternative).
- An alternative is 'applicable' iff its first rule call succeeds.
- An alternative is 'called' by calling its other rules in textual order as long as these rule calls succeed.
- An alternative 'succeeds' iff all of its rule calls succeed.
- A call to a production rule R 'succeeds' iff R has an applicable alternative that succeeds.
- A call to a primitive rule may succeed or fail depending on the prevailing conditions.

Moreover, we have an error condition:

- if any applicable alternative fails, the input does not belong to the generated language (i.e., if an alternative is applicable it is the correct one).

We want to take over these rules as much as possible. In an affix grammar the 'first affix expressions' in the alternatives of a rule may involve primitive predicates, more than one of which may succeed. This problem is (partly) solved by deciding to try them in textual order. With some other modifications this leads us to the flow-of-control rules of ALEPH:

ALEPH rules:

- Execute the affix form in the **root**; it must succeed.
- An affix form is 'executed' by trying in order the alternatives in the RHS of its rule for applicability and executing the first applicable alternative, if any.
- An alternative is 'applicable' iff its first affix form succeeds.
- An alternative is 'executed' by executing its other affix forms in textual order as long as these affix forms succeed.
- An alternative 'succeeds' iff all of its affix forms succeed.
- An affix form which calls a (global) rule R 'succeeds' iff R has an applicable alternative and the executed alternative succeeds.
- An affix form which calls an external rule E may succeed or fail depending on the prevailing conditions.

These flow-of-control rules allow us to view the first affix form as an 'entrance key': one enters the first alternative to which one has the right key. Once this alternative has been entered no others can be reached anymore. An important consequence is that there is only one way to reach a given affix form. This leads immediately to the Central Theorem of ALEPH:

> When the N-th affix form in the M-th alternative is reached, the entrance keys of alternatives 1 through $M-1$ have failed, and affix forms 1 through $N-1$ in this alternative have succeeded.

This Central Theorem is a great help in deriving assertions (see below).

We still have to investigate the error condition inherited from the LL(1) flow-of-control rules; we shall postpone this until 3.3.2.5.

The above rules are (almost) all the flow-of-control ALEPH has: there are no

STELLINGEN behorende bij het proefschrift

On the Design of ALEPH

door

Dick Grune

1. Er zijn drie fundamenteel verschillende manieren om context-restricties in een vW-grammatica tot uitdrukking te brengen.

2. ALEPH-achtige stacks vormen een bruikbaar en goedkoop alternatief voor ALGOL 68-achtige heap-generatoren.

3. Zij A een geordend array van N elementen. De operaties 'zoek een element in A op', 'voeg een element op een gegeven plaats in A tussen' en 'verwijder een element van een gegeven plaats in A' kunnen uitgevoerd worden in een tijd $O(\ln(N))$, $O(1)$ en $O(1)$, als er in het array een (kleine) hoeveelheid loze elementen opgenomen wordt. Het geheugenbeslag van het array is kleiner dan dat van de overeenkomstige (binaire) boom.
   D. Grune, Choosing a Tag-list Algorithm for a Compiler, with Special Application to the ALEPH Compiler, Software — Practice & Experience *9*, 575-593, 1979; *also* IW 89/77, Mathematical Centre, Amsterdam, 1977.

4. Er bestaat een eenvoudige algoritme om de terminale producties van een vW-grammatica te genereren; voor een grote klasse van grammatica's is deze algoritme economisch.

5. De specificatie van mogelijke opeenvolgingen van records op een Standard Labelled Tape zou in grammaticale vorm beknopter, overzichtelijker en implementeerbaarder zijn geweest.
   Standard ECMA-13 for Magnetic Tape Labelling and File Structure for Information Interchange, ECMA, Geneva, 1973.

6. De beperkte blik die een eindstation de programmeur op zijn programma toestaat kan dwingen tot gestructureerd programmeren.

7. Gezien de omvang van op de informatica betrekking hebbende proefschriften en tijdschriften is het noodzakelijk zich te bezinnen op effectievere rapporteringsvormen in dit vak.

8. De verwarring die er bestaat over de vraag of een stack van beneden naar boven groeit of omgekeerd, kan vermeden worden door de stack horizontaal af te beelden.

9. Een auto die voor elke N kilometer afgelegde weg 1 liter brandstof gebruikt ("één op N rijdt"), consumeert al rijdende een brandstofdraad met een doorsnede van $1 / N$ mm^2.

10. Het verdient aanbeveling de twee richtingen van autowegen en buslijnen van verschillende (maar verwante) nummers te voorzien.

11. Het is opmerkelijk dat het onderscheid tussen 'een groot man' en 'een grote man' zich voor de vrouwelijke versie in het Nederlands niet laat uitdrukken.

CASE-, WHILE-, DO-, REPEAT-, UNTIL-, or EXIT-clauses. Rather than emphasizing repetition, ALEPH emphasizes decomposition: each problem is decomposed into several alternatives with entrance keys and each alternative is decomposed into a sequence of sub-problems (which may, of course, be congruent to the original problem). In short, every problem is attacked by recursive descent: ALEPH encourages structured programming in the traditional sense.

```
find name + >name + >list + entry>:
    is empty + list, insert + name + list + entry;
    is name on top + name + list, top of + list + entry;
    next of + list + list1, find name + name + list1 + entry.

    $ approximate declarations of the rules used:

is empty + >list:
    $ succeeds if 'list' refers to an empty list.

insert + >name + >list + entry>:
    $ insert the name in 'list' and put its position in 'entry'.

is name on top + >name + >list:
    $ succeeds if the topmost name on 'list' equals 'name'.

top of + >list + entry>:
    $ put the position of the top of 'list' in 'entry'.

next of + >list + list1>:
    $ put the position of the next element of 'list' in 'list1'.
```

Fig. 8.

One problem associated with structured programming can be solved elegantly in ALEPH: the multi-exit loop. A good example is searching a list for a given name; the search process stops in one of two ways: the list is empty, or we found the name. In the first case we want to insert the name, in the second we are satisfied with the reference to it. Traditionally we would need a multi-exit loop or a global toggle; or we would have to perform the same test twice. In ALEPH we simply state the alternatives and tell what to do; see Figure 8.

It should be noted that, in theory, nothing prevents the programmer from using the same technique in, say, ALGOL 68; the efficiency of the procedure-calling and parameter-passing mechanisms, however, may make the choice less attractive than in ALEPH.

3.3.2.4. Success/failure

We have assumed in the above that any rule can fail (but we have not based any conclusions on that). It soon becomes clear, however, that some rules cannot fail; there are four sources of non-failure:

- an external has an output affix D and its associated function is such that it can always be satisfied by a correct choice of D (e.g., *set to zero*);
- a rule produces ϵ;
- the rule is *produce* (3.3.2.2);
- a rule has an alternative consisting entirely of affix forms which cannot fail.

Through the last property the non-failure propagates through the text: since *where rd is zero* cannot fail, *read* cannot fail, etc.

The Central Theorem shows us immediately that if any alternative but the last one in a rule body has an entrance key that cannot fail, part of the RHS is inaccessible.

3.3.2.5. Side effects

It is the error condition for LL(1)-parsing in 3.3.2.3 which allows us to avoid backtracking, in the following way. When a rule call fails, it has only called other rules that failed. Now since the only terminal rule is *absorb*, and since *absorb* has no side effect when it fails (3.3.2.2), no rule call that fails will have had side effects (by induction). So nothing is modified on failure, and no backtracking is necessary. This is the 'No cure — no pay' principle: one may order something, but if one does not get it, one does not pay.

We would certainly like to carry this nice feature of LL(1) parsing over into our programming language. This is done trivially by forbidding any applicable alternative to fail (either statically or dynamically). But we can do better than this.

Where a CF grammar only has rules (which have side effects on success), ALEPH has rules (which also have side effects on success) *and* primitive predicates (which never have side effects). Moreover, some of the ALEPH rules derive entirely from primitive predicates (3.3.2.2). So in ALEPH a successful affix form does not necessarily imply side effects.

Consequently it is perfectly safe to allow failure of an applicable alternative, provided no affix form with side effects has yet succeeded in the alternative.

Under this regime the 'No cure — no pay' principle holds:

If an affix form (= rule call) fails it has had no side effects.

This means that one can always ask for a service; if it cannot be rendered the request fails and it is as if nothing had happened. The price for this is, of course, a (compiler-checked) restriction on global side effects.

In 3.3.2.4 we have divided the rules into two groups, those that can fail and those that cannot. Now we have a second division, in those that can have side effects (on success) and those that cannot. These divisions are independent, so four classes (rule types) result:

	can fail	cannot fail
can have side effects	PREDICATE	ACTION
cannot have side effects	QUESTION	FUNCTION

(A rule that can neither fail nor have side effects is still useful if it has output affixes.)

Note: the word 'PREDICATE' as a rule type has nothing to do with the word 'predicate' in 'primitive predicate'.

The above classification allows us to give a proper place to *absorb* and *produce*: their rule types are EXTERNAL PREDICATE and EXTERNAL ACTION, respectively. It should be noted that all side effects treated here originate from these two rules. We shall call these side effects 'external', as opposed to the 'global' side effects we shall encounter in 3.3.4.

In principle the compiler could assess these properties, but it is much more useful to have the programmer specify his intentions (opinions) and have the compiler check them. The non-trivial redundancy (3.2.1.2) thus obtained is used for error detection.

Our program is shown in Figure 9; affixes are from now on written in small letters.

*P*4:

```
ROOT text.

EXTERNAL
    FUNCTION set to plus one + >n + n1> = "INCR",
    FUNCTION set + >n + n1> = "SET",
    FUNCTION set to minus one + >n + n1> = "DECR",
    QUESTION equal + >n + >n1 = "EQUAL",
    PREDICATE absorb + >abc = "ABS",
    ACTION produce + >abc = "PROD".

ACTION text: read + n + /a/, print + n + /b/, print + n + /c/.

ACTION read + n> + >abc:
    read symbol + abc, read + n1 + abc,
        where rd plus one is + n1 + n;
    where rd is zero + n.
PREDICATE read symbol + >abc: absorb + abc.
FUNCTION where rd plus one is + >n + n1>:
    set to plus one + n + n1.
FUNCTION where rd is zero + n>: set + 0 + n.

ACTION print + >n + >abc:
    where pt is zero + n;
    print symbol + abc, where pt is minus one + n1 + n,
        print + n1 + abc.
ACTION print symbol + >abc: produce + abc.
FUNCTION where pt is minus one + n> + >n1:
    set to minus one + n1 + n.
QUESTION where pt is zero + >n: equal + n + 0.

END
```

Fig. 9.

We see the impact the rule type classification has on the program: for each rule it is locally clear what to expect of it in terms of flow-of-control. The consistency of the indications is checked by the compiler; here we have strong type checking, not for data types but for rule types (algorithm types).

As with strong type checking on data the errors detected originate from inconsistencies on behalf of the programmer. Suppose there is a rule *xyz* which has ϵ as one of its alternatives and which is used for testing the presence of an *xyz*. Now, if *xyz* is declared as a PREDICATE, the empty alternative will cause an error message, and if it is declared as an ACTION, its use as a test will be noticed.

For an application of this type checking in the construction of a program, see 4.3.3.

3.3.2.5.1. Overriding the consistency check

The above works fine for a problem from which all backtrack has been removed, but it effectively prevents the programmer from programming his own backtracking. This situation is felt to be too restrictive. There are some legitimate reasons for a programmer to want a failing rule to have side effects, e.g.:

- during debugging it may be necessary to trace the activities of a rule even if it ultimately fails;
- the input grammar is not completely LL(1), i.e., at a few points the parser has to peek ahead (such a grammar can sometimes be much simpler than a pure LL(1) grammar for the same language).

We shall therefore allow failure after side effects, but only under protest: the compiler gives a warning message (ALEPH Manual 3.2.2.b). Normally this serves as an error message and the programmer can easily mend the situation.

3.3.3. Affixes

Rules in an affix grammar can have bound affixes (those that occur in the LHS and in the RHS) and free affixes (that occur in the RHS only). In ALEPH these are termed formal and local affixes, or 'formals' and 'locals'. To avoid errors we shall require the locals to be declared in the LHS as well; they will be distinguished from the formals by a preceding − (minus-sign).

The 'control' of an affix grammar (S) contains information about the nature of the bound affixes (=formals) of a rule. They can be 'inherited' or 'derived', corresponding in ALEPH to 'input' and 'output' formals, respectively. An input formal has a value upon entry to the rule (is 'initialized'), an output formal must have received a value when the rule ends.

Of course it is necessary that the input affixes of an affix form have all obtained a value (are 'initialized') when the affix form is executed. Now, since

- the Central Theorem states that there is only one path from rule entrance to a given affix form, and the C.T. gives that path,
- the initial states of all formals and locals at rule entrance are known from the LHS, and
- for each affix form A on the path the effect on the actual affixes passed to it is known from the LHS of A,
  the compiler can ascertain in an efficient way that the value of an affix will not be used before that affix has received a value. No run-time checking is

necessary. A similar test can ensure that an output formal will always receive a value.

The details of this test depend on the affix-passing mechanism.

3.3.3.1. The affix-passing mechanism

The affix-passing mechanism has to obey two conditions: the value of an inherited affix must be available inside the rule, and the value obtained by a derived affix inside the rule must be made available to the caller.

If we do not allow the value of an affix to be changed (once it has obtained a value), then the story ends here: all affix-passing mechanisms which conform to the above conditions are indistinguishable (except, perhaps, as to efficiency).

At the time of the design, however, we did not seriously consider the possibility of programming with initializable constants only, and felt that variables were indispensable. However debatable this decision may be (3.6), it has led to an interesting extension of the 'No cure — no pay' principle to local variables.

Since rules need the possibility to change values of affixes of calling rules, it seems that we need at least call-by-reference (or a more general mechanism). Call-by-reference, however, can surprise the programmer painfully with invisible aliases, as in:

ACTION produce a or b + p> + q>:
set + p + /a/, set + q + /b/, produce + p.

where a call *produce a or b + x + x* produces '*b*'. Moreover, backtrack rears its ugly head again when a rule fails after having changed the value of an (output) affix.

On the other hand it is clear that call-by-value alone is insufficient.

A good in-between is found in 'copy-restore': upon rule entry all input affixes are copied to a local work space, and upon rule exit all output affixes are restored from that local work space. If we now suppress the restoring if the rule fails ('copy-*maybe*-restore'), no effects on affixes will propagate upwards upon failure, and a failing rule will never spoil information: the 'No cure — no pay' principle also holds for affixes.

Under these circumstances we can easily introduce 'in-out-affixes', which must have a value upon entrance and which return the (possibly changed) value; notation: *+ >tag>* .

The copy-maybe-restore mechanism allows us to view the (formal and local) affixes as local variables, some of which are already initialized upon rule entrance and some of will be returned to the caller if and when the rule succeeds. This mechanism is easy to explain and efficient to implement. It aids programming in that it supplies automatic backtracking on local variables.

The introduction of variables allows a shorter form of our program, as given in Figure 10.

3.3.4. Globals

ALEPH is intended for the writing of fair-sized programs like compilers, text justifiers, etc. With such programs it often happens that a rule at the periphery of the directed graph (3.3.2.1) needs a piece of information which has to retain its value to the next call of that rule. Examples are the line number and page heading for a print rule, and the name list (identifier table) in a parser rule which handles identifiers.

P5:

```
ROOT text.

EXTERNAL
    FUNCTION increment by one + >n> = "INCR",
    FUNCTION set + >n + n1> = "SET",
    FUNCTION decrement by one + >n> = "DECR",
    QUESTION equal + >n + >n1 = "EQUAL",
    PREDICATE absorb + >abc = "ABS",
    ACTION produce + >abc = "PROD".

ACTION text - n:
    read + n + /a/, print + n + /b/, print + n + /c/.

ACTION read + n> + >abc:
    read symbol + abc, read + n + abc,
        where rd plus one + n;
    where rd is zero + n.
PREDICATE read symbol + >abc: absorb + abc.
FUNCTION where rd plus one + >n>: increment by one + n.
FUNCTION where rd is zero + n>: set + 0 + n.

ACTION print + >n + >abc:
    where pt is zero + n;
    print symbol + abc, where pt minus one + n,
        print + n + abc.
ACTION print symbol + >abc: produce + abc.
FUNCTION where pt minus one + >n>: decrement by one + n.
QUESTION where pt is zero + >n: equal + n + 0.

END
```

Fig. 10.

VW-grammars and affix grammars accommodate these entities by aggregating them in metanotions or affixes and passing them up and down all rules concerned. The **NEST** in the formal grammar of ALGOL 68 [VAN WIJNGAARDEN 75] is a good example.

From a practical point of view there are two objections to this technique. Given the affix-passing mechanism explained above (3.3.3.1) it results in massive copying and restoring of large data structures; and it forces the programmer to specify long tails of affixes to his rules.

The latter problem can be obviated by taking many (disparate) affixes together in a single affix which is then passed to all rules concerned. It is clear that we loose structure this way: many rules get access to affixes they do not really need.

Once we have lumped into one affix all affixes in which there is more than local interest, we can (partly) solve the former problem: make that affix implicitly accessible to all rules. In fact we have reinvented global variables. Of course this solution comes

at a price: we loose the automatic backtracking which we had when all affixes were still local (but we keep it for those that remain local).

Fortunately this solution does not really create a new problem. We already had rules which have (external) side effects because they absorb input or produce output. Now we also have rules that have (global) side effects because they modify global data. The same criteria for backtracking hold (see in particular 3.3.2.5.1).

A special case is the modification of global data through output affixes of a FUNCTION or QUESTION. Thus an affix form can have side effects, even if the called rule cannot. All this is covered in ALEPH Manual 3.9.1.

The introduction of globals allows us to relieve *absorb* and *produce* of their exception status. All input and output in ALEPH is done through files, and, in the case of character I/O, through 'charfiles'. Notation:

CHARFILE input = >"INPUT", output = "OUTPUT">.

Note the use of the right-symbol > ; placed in front it indicates that the file has been prefilled, placed behind it indicates that the file will be passed back. Now *absorb* and *produce* just correspond to two externals which receive a file as an affix:

```
EXTERNAL
    PREDICATE get char + ""file + char> = "GETC",
    ACTION put char + ""file + >char = "PUTC".
```

The difference between global and external side effects has vanished.

3.3.5. Affix rules

The affix rules of an affix grammar correspond to data types in a programming language. Although much can be said about the realization of those data types, we shall not pursue this subject any further in this thesis. The actual decisions in ALEPH, especially with respect to data-aggregating mechanisms, will be explained in 3.5.

3.3.6. The final program

Given suitable external routines INCR ... PROD, program *P5* is an executable ALEPH program. A number of externals, however, have been predefined in ALEPH, and it is good practice to restrict oneself to these. Since user-declared externals are not automatically portable they should be used for exceptional purposes only.

INCR and DECR are predefined and called *incr* and *decr*. There is a special notation for setting a variable to a given value:

$0 \rightarrow n$

and, likewise, equality can be tested by

$n = 0$.

produce is handled by *put char* (3.3.4); *get char* behaves like *absorb*, but only partially so. *get char* yields the next character from the file and fails on end-of-file (it would be unpleasant to have to find out what the next character was by using *absorb* alone!). So we have to rewrite *absorb*, using a global VARIABLE.

The final form of the program is given in Figure 11 (comment behind $s).

P 6:

```
ROOT text.

CHARFILE input = >"input", output = "output"<.
VARIABLE char = / /.                    $ some suitable initialization.
CONSTANT end of file = max char + 1.    $ 'max char' is predefined.

PREDICATE absorb + >abc:
    char = abc, get next char.
ACTION get next char:
    get char + input + char;            $ if it is there.
    end of file → char.                 $ otherwise.

ACTION text − n: get next char:         $ the real initialization.
    read + n + /a/, print + n + /b/, print + n + /c/.

ACTION read + n> + >abc:
    absorb + abc, read + n + abc, incr + n;
    0 → n.

ACTION print + >n + >abc:
    n = 0;
    put char + output + abc, decr + n, print + n + abc.

END
```

Fig. 11.

3.3.7. The notation

A few words about the notation are in order. There has been strong pressure from prospective users against the use of pluses as affixers in favour of a notation with parentheses and commas, as in the ALGOLs, Pascal, etc. We have resisted this pressure, mainly because it was directly connected with the wish to write (nested) expressions as affixes. The values of these expressions, however, have to come from rule calls, which may fail. The idea clearly runs contrary to the philosophy of ALEPH, where the 'value' a rule returns is its success or failure and where computational results are passed on as affixes.

It should also be noted that the ability to return computational values is only a partial blessing: as soon as a procedure returns more than one result, the programmer has to resort to, possibly legalized, trickery. A good example is the integer division which naturally yields both quotient and remainder. No major language of today makes both results simultaneously available (but see *divrem*, ALEPH Manual 5.2.1).

3.3.8. Conclusion

We have shown that by exploiting the analogy between grammars and programs, and between parsing and problem solving, a practical language can be designed that has some properties not generally found in programming languages.

Among these properties are:

- a simple and effective flow-of-control based solely on selection, decomposition and procedure calling;
- a Central Theorem which states in simple terms the conditions which apply when a given construct is reached;
- an efficient compile-time check on the initialization of variables;
- a firm and compiler-checkable concept of side effects.

A few other features indispensable to a modern programming language, like exception handling, modularization or a programming environment, do not follow directly from this analogy. For the development of CDL2 in this direction see [BAYER et al. 81].

3.4. The portability of ALEPH programs

ALEPH is, in essence, a very simple language. Broadly speaking, its basic building actions are:

- pass parameter,
- call subroutine, and
- jump conditionally on boolean result,

which can all be implemented with reasonable ease on any reasonable machine. During the design of ALEPH care has been taken not to spoil this simplicity, and with that the machine-independence and portability, more than necessary. As a result of this, most of the portability problems listed in [TANENBAUM, KLINT & BÖHM 77] cannot occur in an ALEPH program. Nevertheless there are some obstacles which will or may have to be faced by the programmer who attempts to transport an ALEPH program from a source machine to a target machine. In the order presented in [TANENBAUM, KLINT & BÖHM 77] they are:

1. ALEPH may not be available on the target machine.
2. The program may use 'user-externals' (ALEPH Manual 5.1) or local 'pragmats'.
3. The program may rely on numerical values of the character set.
4. The target machine implementation may have more restrictive overflow conditions.
5. The target machine implementation will have a different idea about the contents of the **string-denotation** in a **file-description**.
6. The program may generate machine-dependent output, even if it is itself machine-independent (i.e., besides being portable, a program should be retargetable).
7. If two or more co-operating ALEPH programs are to be transported, they may run into communication problems.

All the problems apply a fortiori to the ALEPH program we are concerned with here, i.e., the ALEPH compiler. We shall now consider each of these problems in turn, both for the ALEPH compiler and for the general ALEPH program.

3.4.1. ALEPH may not be available

This problem applies only to the compiler. As explained in 4.2, its solution is supported by the use of ALICE and by bootstrapping (4.5).

3.4.2. User-externals and local pragmats

Neither user-externals nor local pragmats should occur in portable software. Care has been taken in the design of the compiler to avoid algorithms which would make user-externals desirable. For example, hashing methods for the identifier-list algorithm have been rejected, since calculating hash values efficiently in a machine-independent way is difficult, because of overflow problems and limitations in the data access. See [GRUNE 77].

3.4.3. Numerical values of the characters

The bit patterns, and thereby the numerical values, assigned to characters are generally machine-dependent. The use of a user-external to obtain such values efficiently is undesirable, as indicated above. This problem can mostly be avoided by using **character-denotations** whenever possible. If, for efficiency reasons, it is desirable to use characters as indices in indexing a fixed array (as it is in the ALEPH compiler), the contents of the array can be written in a code-independent way using **character-denotations** and then be reordered at run time so as to fit the collating sequence of the actual character code. For details see [VAN DIJK 82].

3.4.4. More restrictive overflow conditions

In general, an ALEPH program may run into overflow problems in one of three ways: an arithmetic operation may generate a result outside the integer capacity; the program may run out of memory space; and a stack may run out of virtual address space (an overflow condition specific to ALEPH). All three are a definite threat to portability.

Part of the integer overflow problem is alleviated by the arithmetic operations of ALICE: the compiler need not do any arithmetic and can delegate all of it to ALICE in the form of **calculations** (ALICE Manual 3.1.1). It should be noted that, regardless of overflow conditions, the compiler *has* to delegate some of it to ALICE, since it does not know various implementation-dependent values like *max char*, *int size*, *min addr*, etc.

This does not mean that arbitrarily large results can be obtained; if a result gets too large, an ALICE **calculation** will detect the overflow.

In the compiler design arithmetic has been restricted to the bare minimum, and care has been taken to ensure that results will remain less than 2^{15}. It is clear that ALEPH will not run reasonably on a machine with smaller integers anyway, for lack of virtual addressing space. (This implies that the machine realized by ALICE has to use at least two bytes for modelling integers on byte-oriented target machines.)

The compiler is very careful about memory usage. Any stream of information which is produced sequentially and consulted sequentially is kept in a file rather than in a stack.

Memory requirements could be lowered still further by putting the direct-access information in secondary memory through a background-pragmat (ALEPH Manual

6.1), but this solution is not very practical in porting the compiler, since the background-pragmat will probably not be one of the first features to be implemented at the target site.

The amount of virtual address space available to a stack can be controlled by the **relative-size** in its **stack-description**. It can be adjusted to the local situation, but only *after* the compiler has been installed. Therefore, the **relative-size**s in the distributed compiler are adjusted to the amounts of virtual address space needed for the compilation of the compiler itself. This can, however, be done only approximately, since, e.g., the virtual address space occupied by strings is implementation-dependent. The **relative-size**s are based on a string packing of one character per word.

3.4.5. Strings in file-descriptions

The nature and amount of the information a program has to know about the files it uses differs greatly from operating system to operating system [NOS/BE 79, RITCHIE & THOMPSON 74]. The most universal properties can be specified in a machine-independent way in ALEPH. These are whether the file is to be read or to be written, and whether it contains characters or integers (ALEPH Manual 4.2).

Further information can be supplied in a string; this string is passed unmodified to ALICE in a **file-administration** macro sequence (ALICE Manual 3.2.3.1). The contents will be installation-dependent; on the Cyber, e.g., it contains the file name, an indication whether the file contains printer control characters, an indication whether the file is allowed to reside on magnetic tape and some information on how the file name can be changed upon program invocation. The receiver has several options here:

- he can adapt his (first version of the) ALICE processor to this convention,
- he can change the strings in the ALICE file (they are easy to find),
- he can take the file name to be the name of a data-description in the operating system, if his operating system works that way.

It should be noted that the problem of machine-independent file identification is especially serious in compilers. Many portable user programs need only a standard input file and a standard output file, as they are predefined, e.g., in ALGOL 68 [VAN WIJNGAARDEN 75] or C [KERNIGHAN & RITCHIE 78]. A compiler, however, will need scratch files, libraries, several output files, etc.

3.4.6. Machine-dependent output

There are several situations in which a program which is by itself machine-independent produces machine-dependent output. Examples are compilers and graphic display systems. Porting such programs can be simplified by introducing a machine-independent problem-oriented interface. All output of the program is formulated in terms of this machine-independent interface, thus enabling the program to be portable. The output is then passed to a (hopefully simple) post-processor that converts it into machine-usable form. In the case of our ALEPH compiler the interface is provided by ALICE.

We shall not address the problem of machine-dependent input here.

3.4.7. The need for job control

A job-control language is used to describe the general logistics of a job: the origin of input files, the destination of output files, the sequence of programs to be called, etc. It will be used extensively in the bootstrapping process described in 4.5, and the receiver is expected to be reasonably proficient at it. It is totally different for different operating systems, so the best a writer of portable software can do is to minimize the requirements.

The minimal requirements in the case of the compiler are:

- there is one input file: the ALEPH source text,
- there are two output files: the listing and the ALICE code,
- there is one program: the compiler itself.

Such an arrangement, however, would mean that all external declarations must be built-in and all intermediate results kept in memory: memory requirements would become appalling. Therefore, the external declarations are kept on a second input file and are read as necessary; several scratch files are used.

The compiler is distributed as one program. As explained in 4.3, the ALEPH compiler is not really an N-pass compiler; rather, there is an information-collecting phase, which fills stacks and files, followed by a number of information-processing phases, which produce the various parts of the ALICE code from this information. If memory shortage requires so, some of these phases can be split off into a second separate program. The pertinent information will then have to be passed on by means of ALEPH 'datafiles'.

3.5. Data structures in ALEPH

Data structures present themselves in the design of ALEPH in a natural way as affixes. In principle each affix comes with a grammar which produces all 'values' the affix may take. Such a value is passed around from rule to rule and is finally handed to an external rule (a 'primitive predicate' of the affix grammar) that may operate on it. The external rule may create new values and succeed or it may fail. The programmer should be able to specify the internal structure of such a rule.

The first thing an external rule will in general do is to take apart the affix value, i.e., to parse it. For that, however, the control structure of a normal ALEPH rule is quite adequate and we don't need the escape mechanism of an 'external rule'. Likewise, new affix values can be created through normal ALEPH rule calls.

If we can use the terminal symbols of the affix grammar (the set A_t in $AG1$ in 3.3.1.3) as constants, the only 'external rules' we need are comparison and copying, plus a storing and addressing scheme. These are provided as the ALEPH primitives **identity**, **transport**, **extension** and **element**.

This set may be sufficient in theory, but it is not efficiently usable: we are reduced to doing unary arithmetic (which should not amaze us, since it is the same with affix grammars!).

In ALEPH as it stands now the only basic data type is the *integer*. Names can be given to integer constants, integer variables, lists of integer constants and lists of integer variables, through the following language constructs:

- **constant-descriptions**, which give names to (compile-time) constants;

- **variable-descriptions**, which declare global integer variables; initialization with a compile-time constant is obligatory;
- **table-descriptions**, which declare "tables" of integer constants, the "elements"; the elements are grouped in "blocks", a block is indexed by an integer (a "pointer") and an element is selected from a block through a named "selector";
- **stack-descriptions**, which declare "stacks" of integer variables; a "stack" is like a table, but the values it contains may be replaced and blocks may dynamically be added to or removed from the right end (see 3.5.1).

There are two more language constructs to facilitate data handling:

- **string-denotations**; strings can only reside in tables and stacks where they appear as lists of integers in a machine-dependent format;
- **file-descriptions**, which provide communication channels with the world. Integers pass through them, interpreted either as integers or as characters. There is a special way to send pointers to another program (ALEPH Manual 4.2.2).

Data items can be handled either by means of the four ALEPH primitives mentioned above, or through the standard external rules available to the user; for the latter see ALEPH Manual 5.

3.5.1. Stacks

A flexible information storing device is an important facility for a compiler writer, or for the programmer of any fair-size program, who has to cope with accumulating information of unpredictable size.

Fixed-size arrays, still often used in compilers and editors, use memory inefficiently and tend to be too small at inconvenient moments. Linked lists are better, but need room for the links, provide no direct access and have deallocation problems.

The ALEPH 'stack' can be viewed as an extensible array of blocks of elements (integers). A block can be reached by indexing with a pointer and an element in a block can be reached by selecting with a name. The right end can be used in stack-fashion: a block can be pushed onto it through an **extension** and the right-most block can be removed through a call of *unstack*. Single elements cannot be pushed on a stack (unless the stack is defined so that one element constitutes a block).

The integers used for indexing a stack are chosen by the system, in such a way that they identify the stack they belong to. The programmer can use a **pointer-initialization** or a **limit** to get hold of such a value once a block has been added to the stack and the standard-external *was* allows him to check whether a given integer value is a valid index to a given stack.

There are no direct limitations to the size of a stack. The collection of stacks in a program may grow as far as the operating system allows. Since the system may conceivably run out of integer values to be used as indices, very large stacks may cause problems (ALEPH Manual 4.1.4).

For the programmer stacks are about as convenient as heap-generators in ALGOL 68: on the one hand one has to be more careful about deallocation, but on the other hand they allow direct access. The run-time efficiency of stacks, however, is much greater than that of heap-generators. The latter require a garbage collector whereas the former need a simple shifting algorithm only. Implementation note: the rule-call stack is treated internally as a normal ALEPH stack.

In the present implementation the contents of the stacks lie in a contiguous piece of memory. If the extension of one stack causes it to bump into the next, the available space (possibly increased by a systems call) is redistributed by shifting the contents.

A disadvantage is that since all indexing is done with integers, each access to an element has to go through the administration block of its stack.

Only *global* stacks are available. There are no fundamental difficulties with *local* stacks, but there is no syntax for them. Stacks can, however, be passed on as parameters.

3.6. Evaluation of some compromises

In the design of ALEPH two major compromises have been made: the introduction of variables and that of **compound-member**s.

The original design left us with data items that are declared, receive a value once and remain unaltered until the end of the declaration range. If the environment needs modification a new range must be opened, a new data item must be declared and it must be set to the modified value by passing it as a inherited affix to an appropriate rule. Now this is fairly acceptable for small data items, but it is hard to implement efficiently for large data structures like name-lists, etc. Furthermore this approach causes a considerable growth of the run-time stack. On the other hand, all these problems yield to optimization techniques, especially in ALEPH, where the flow-of-control is very much restricted.

In total, the introduction of variables has probably improved the language more in usability than it has damaged it in complexity (see, however, [WULF & SHAW 73]).

The introduction of **compound-member**s was a matter of convenience for the programmer. It is only slightly more work to write a separate **rule-declaration** for every **compound-member**, but the main burden comes from the need for meaningful names. On the other hand, the existence of **compound-member**s has created big problems, as there are:

- the 'spoil and fail' effect, which necessitates the insertion of hidden locals;
- the determination of the rule type of a **compound-member**.

Both the language and the compiler would have been simpler without **compound-member**s, probably without great detriment to its usability. The introduction of **compound-member**s is slightly regretted.

Of the minor compromises, two will be mentioned here: the introduction of the **classification** and the decision that the arithmetic operations be FUNCTIONs rather than QUESTIONs.

A **classification** (ALEPH Manual 3.8) looks like, and performs functions similar to, an **alternative-series** (ALEPH Manual 3.2.2), except that the selection of the **alternative** is done by sequentially comparing the value of a variable to a number of constant ranges rather than by sequentially trying 'entrance keys'. Although essentially superfluous, it is a well-known language feature (CASE, SWITCH and the like) that helps the programmer in expressing the concept of obtaining an action by indexing, and helps the implementer in optimizing the code. It can be implemented without undue difficulty and is responsible for 6 of the 83 ALICE instructions (4.4.1).

For some time we have played with the idea that, e.g., the *plus* on integers is in essence a request rather than an order, since the result may not exist in a given

implementation due to integer overflow, and consequently *plus* should be declared as a QUESTION (see [BOSCH, GRUNE & MEERTENS 73, 3.3]). Likewise, accessing an indexed element of a list should be considered a request rather than an order, since the indexed element may not exist. Because of the flow-of-control rules of ALEPH (3.3.2.3) this would force the programmer to supply alternatives for the case that the request failed. Ultimately the only run-time error messages from any ALEPH program would be 'Memory resources exhausted' and 'Allotted time exceeded'.

However attractive this concept may be, the problem is that the user *cannot* generally supply a reasonable alternative if the result of an arithmetic operation has no representation on his machine, except to abort the program (see, however, 3.3.8). In the case of the indexed element he *will* not supply an alternative since through using the index he has shown his conviction of its appropriateness, which, if he had doubted it, he could have verified through a call of *was* (ALEPH Manual 5.2.4).

4. ON THE DESIGN OF THE ALEPH COMPILER

4.1. History of the compilers

The first CDL translators, written by C.H.A. Koster, were combinations of transducers and macro processors, which transformed the input text (in CDL) piecemeal into output text (in ALGOL 60). Hardly any context checking was done at this stage, nor was it really necessary since the subsequent ALGOL 60 translation would catch most errors (but since the ALGOL 60 translator was operating on the wrong level, diagnostics left much to be desired). Some syntax checking was provided, since the translator was driven by the grammar of CDL, but context checking in a language that does not restrict the order in which the declared items occur requires an amount of foresight that can only be achieved by a multi-pass process.

Later versions (like the one published in [KOSTER 71a]) introduced some measure of context checking, though remaining one-pass. Information about the use of an identifier was collected, and, when its declaration was met, a consistency test was performed. This collecting of information was done solely for the benefit of the programmer, so as to provide him with early warnings about errors; it played no role in the transformation process itself.

All these versions of the translator ran on the Electrologica EL-X8.

About the same time that ALEPH emerged and the need for an efficient ALEPH compiler arose, the EL-X8 ceased to be available and the project had to be moved to a Control Data Cyber 72. ALGOL 60 on this machine was not well supported. That, and the wish for an efficient compiler, led to the decision to generate COMPASS (the assembler for the Cyber 72) code [COMPASS 79] rather than ALGOL 60.

Thus the first ALEPH compiler, written by R. Bosch, was immediately involved in a fairly complicated cross-bootstrapping process between ALGOL 60 on the EL-X8 and COMPASS on the Cyber. The shock was eased by the use of a set of COMPASS macros that mimicked the primitives needed by ALEPH, thus putting a large part of the burden on the macro processor incorporated in the COMPASS assembler.

If context checking through ALGOL 60 was unsatisfactory, context checking through COMPASS was non-existent. Moreover, the introduction of the the copy-maybe-restore mechanism (3.3.3.1) made context knowledge indispensable, since it needs information about the affixes for its correct translation.

So Bosch modified the compiler to make two passes over the text, do context checking and produce COMPASS directly, thus removing the last reminiscences of a macro processor. It is this compiler that was used to implement the portable ALEPH compiler described in this thesis.

4.2. The design technique

4.2.1. Design criteria

The ALEPH compiler mentioned above has been a workable product on the Control Data Cyber since 1974. However, originated in a turmoil of changing languages and machines in an environment where even the physical transport of files was a problem, it shows all the signs of having grown rather than having been designed.

Since one of the main purposes of ALEPH was to serve as a vehicle for portable compilers, its portability was of great importance. Now, the old compiler was written

with only one purpose in mind, to get ALEPH running. It was deemed impossible to convert it into a portable compiler.

The design of the new compiler focuses on two issues:

- portability,
- minimal memory requirements (equally important for portability).

In 3.4 it is shown that an ALEPH program in general is fairly machine-independent. But if that program is a compiler we run into a specific problem: the machine-independence of the generated code ('retargetability'). The approach to its solution is explained below. For the minimal memory requirements, see 4.3.1.

An additional requirement was that the design technique should be so simple and effective, that the design could be done by a single person. This resulted in the factorization of the design as explained in 4.2.3.

4.2.2. The portability of the compiler

The machine-independence of CDL was based on the idea that the compiler should be given, in addition to the program to be compiled, a description of the target machine in some formalized form. The compiler would then turn out object code tailored to the target machine.

The CDL compiler did this by reading, in a fixed order, pieces of text to be produced for, e.g., 'beginning of procedure', 'jump to label', etc., and consequently a machine description had to be given in these terms. This works well if the machine lends itself to expressing these primitives (and, since that machine was ALGOL 60, it did) and if the changes in the machine are small and superficial (like a change from ALGOL 60 in underline-style to ALGOL 60 in apostrophe-style).

As soon as one wants to compile towards a totally different machine, e.g., an assembler, this scheme breaks down. The required primitives just aren't there. It has been suggested that for target machines of this type the machine description should include items like the number of registers which are available for certain purposes, the properties of the arithmetic used, the alignment requirements for data, etc. [BOURNE, BIRRELL & WALKER 75]. Although a modicum of machine-independence can be reached this way, it turns out that it is difficult to give a correct machine description of this nature. Now, if the compiler and target machine are located close together, repeated corrections of the machine description are a minor nuisance, but if they are far apart this technique gives rise to the proverbial debugging loop across the Atlantic [RICHARDS 77].

In the mid seventies a new concept became popular, the 'machine-independent intermediate code' [BROWN 77]. The idea is that a compiler at site A translates a program into this intermediate code such that the resulting translation is not a grain more machine-dependent than the original. This translation is then shipped to sites B to Z where it should be possible to transform it, with reasonable effort, into something locally usable.

It should be noted that for each program there is only one translation into the machine-independent code, regardless of the actual machine which does the translating. So the whole process could equally well be performed at site K and the (identical) result sent to sites $A \cdots J, L \cdots Z$.

The success of this scheme hinges on the choice of the machine-independent intermediate code. We have two options here: either to use an existing widely available

language or design a new code tailored to our needs. Of the widely available existing languages only FORTRAN is a candidate. It was rejected off-hand because of its obvious draw-backs. In hindsight it may have deserved a better chance than it got. Its draw-backs are indeed obvious: recursion is pretty hard to simulate in FORTRAN, input/output can only be done a line at a time, and dynamic memory management is alien to FORTRAN. Its advantages as an intermediate code are much less obvious: programs using a small, well-chosen subset of FORTRAN are quite portable, there are excellent optimizing compilers for it, and the above problems *can* be solved in a practical way: see [WAITE 75, p.315] for a (partial) solution.

A.P.W. Böhm has studied the problem of designing a machine-independent intermediate code for the specific purpose of implementing ALEPH. This has resulted in ALICE, *AL*eph *I*ntermediate *C*od*E*, which is the code produced by the new ALEPH compiler. ALICE is described in detail in [BÖHM 77]; the reader can find a short introduction in 4.4.1 in this thesis. Böhm has written a pilot implementation of ALICE on the PDP11/45 under UNIX [RITCHIE & THOMPSON 74]; it is described in [BÖHM 78].

ALICE is a very clean interface and through its cleanness has been a great help in structuring the design and implementation of the compiler. It is doubtful if FORTRAN could have rendered a similar service.

The following paragraph treats the role that ALICE has played in the design of the compiler.

4.2.2.1. ALICE as a target code

Prime concern in the design of ALICE has been the ease of implementation on a variety of machines, so that a receiver will, hopefully, have minimal trouble in implementing it on his local machine. Equally important, but needing less emphasis, was its suitability for expressing the semantics of ALEPH. Concern for the ease of translating ALEPH into ALICE code, however, came only third. In the design of ALICE simplicity (and versatility) of translation has always prevailed over simplicity of generation. One reason for this is that translation must be done for each machine type on which ALEPH is to be installed, whereas generation needs to be done only once. As a consequence ALICE is a peculiar machine for which it is not particularly easy to generate code.

The actual situation is not as bad as it sounds. ALICE may pose many requirements, it is also well-structured enough that these requirements can easily be localized and dealt with.

One way of localizing all requirements is the bottom-up approach. We start with the ALICE macros as building blocks. Each needs zero or more parameters and supplies zero or more parameters. We then combine these building blocks into larger units, each needing and supplying parameters, until we have a set of building blocks which can support an ALEPH program.

A disadvantage of this method is that the usefulness of a building block becomes apparent at a very late stage only, and one may easily design superfluous building blocks.

In a top-down design, however, one never loses sight of the purpose, since it is the only thing pursued. Here we start from the ALEPH constructs and work our way down along the 'tree of obligations'; for each obligation:

- either we convince ourselves that it is trivial to fulfil,
- or we subdivide it into further obligations.

The crucial point is the subdivision. Each subdivision defines an interface, be it ever so simple, and the art of top-down design is actually the art of choosing interfaces.

To see this process at work we shall show it below (4.2.2.2) in sufficient detail for the ALEPH construct **identity** (i.e., comparison). We shall perform it step by step and shall let ourselves be guided only by the principles of top-down design and the structure of ALICE.

The analysis performed there makes it clear that no (or hardly any) ALICE code can be produced until the entire ALEPH program has been read and digested. ALICE code is then produced from the digested form. This does not necessarily imply two passes over the input; only if the ALICE translation more or less follows a version of the ALEPH source text (as modified by the first pass) can we speak of a 'second pass'. In practice it hardly ever does: the information is collected in stacks, from which the appropriate ALICE code is generated, often in an order which is totally unrelated to the order in the source text.

The strict division between an information-collecting phase and an information-processing phase has the additional advantage that all information for semantical error-detecting and error-reporting is available when it is needed.

4.2.2.2. An example

Suppose we have found in the ALEPH text an **identity** (ALEPH Manual 3.4.1)

xyz = 72

and we want a translation into ALICE. The corresponding ALICE form is a **statement**, which is either a **call**, an **ext-call** or a **primitive** (ALICE Manual 3.3.3). Now a **call** requires an identification of the ALEPH rule to be called, which is missing, and the **primitive**s are of a different nature altogether. So an **ext-call** is indicated, with an **stag** *EQL*. Such an **ext-call** (ALICE Manual 3.3.8.1) requires a description of its input parameters *xyz* and *72* in the form of a **copies-to-input-gate**.

If we assume for the sake of argument that *xyz* is a global variable, its **copy-to-input-gate** amounts to **load-variable-in-v_reg** (the structure of an ALICE program is briefly explained in 4.4.1):

```
LVV repr_of_xyz          $ Load V_reg from Variable
```

But the generation of this **statement** requires a *repr_of_xyz*, which can only come from a preceding ALICE **variable-description**

```
VAR repr_of_xyz,valref,repr_of_next_var,"xyz"
```

This in turn requires a **valref**: the initial value of *xyz*, which must come from some preceding **value-definition** or **calculation**, e.g.,

```
INT valref,0
```

(the second **repr** in the **variable-description** is the **repr** of the next **variable-description**, a complication we are not concerned with at the moment).

So the use of the *LVV*-macro requires the foresight of having already generated a corresponding *VAR*-macro which again requires the foresight of having already

generated an *INT*-macro.

Likewise the translation of the *72* requires an *LVC*-macro which requires a **repr** which must come from a preceding *CSS*-macro and a **valref** which must come from an *INT*-macro preceding both other macros. Moreover, all *INT*-macros of the whole program have to come together (in an ALICE **values**) and so have all *VAR*s and *CSS*s.

Now it might be argued that all these macros could be generated when the need for them becomes apparent, that each could carry an indication of its eventual position in the ALICE-file, and that sorting could finish the job (a technique already used in one of the first FORTRAN compilers [SHERIDAN 59]). For simple cases this works, but the scheme fails already on indexed elements:

*next*list[p]*

will be translated using

```
IIP                     $ Index Input Parameter
LVV repr_of_p           $ Load V_reg from Variable
LAG repr_of_list        $ Load A_reg from Global
LVI number_of_next      $ Load V_reg Indexed
```

The last macro requires the position of the field *next* in a block on the stack *list*, knowledge which can only be obtained from the ALEPH **stack-description** which maybe we have not seen yet.

All this would be simpler if ALICE allowed a construction like

```
IIP
LVV p
LAG list
LVI next
```

The original sequence, however, is easier to translate to machine code and therefore preferred (4.2.2.1).

4.2.3. The four stages of the design

The above observations were made the basis of the design technique. It was clear from the onset that the design technique had to be structured in some way or another, since the design was a one-person project and the complexity of even a relatively small compiler as for ALEPH is too great to allow one person to master all the details all of the time. Moreover, we already possessed an ALEPH compiler designed by accretion and erosion, a design technique (or lack thereof) which had yielded a clumsy compiler, which, though working, was infested with traces of design changes, ad hoc solutions and unexpected machine-dependencies.

The design technique in principle consisted of four stages:

- Stage 1: All ALEPH constructs were considered and for each an ALICE translation was chosen.
- Stage 2: For each ALICE construct (comprehensive constructs as well as macros) a list was made of the information items it needed, in the context in which it occurred.

- Stage 3: Ways were devised to extract this information from the source text, resulting in algorithms which acted as if each were a separate pass over the source text.
- Stage 4: The algorithms were supplied with concrete data representations and combined into a single compiler.

In practice Stage 1 turned out to be almost trivial: because ALICE was specifically designed as an ALEPH intermediate code, the ALICE translation was obvious in all but a few cases. Only when dealing with the typical ALEPH flow-of-control operators like **comma-symbol**, **semicolon-symbol** and **compound-member**, one has to realize that their semantics is expressed in ALICE through the true- and false-addresses. If proper attention is paid to this, Stage 1 can be incorporated in Stage 2.

Stage 2 is performed in top-down fashion. Our first aim is to produce an ALICE **program**, which supplies us with the secondary aims of producing the ALICE items **string**, **status-information**, **values**, **data**, **communication-area** and **rules**, the first of which requires the 'title of the program', etc., etc. A representative part of the process is described in great detail in 5.1.

Stage 3 was performed on the pattern left behind by Stage 2. This pattern consisted of lists of requirements for information to be extracted from the source text and abstract algorithms waiting for further information about their data types. Now that all parts of the ALICE program have been considered once, the details can be filled in. A representative part of Stage 3 is described in 5.2.

It turned out that Stage 4, choosing concrete data types and merging the algorithms into a single compiler, was easily combined with the actual writing of the compiler. The compiler was written by the programmer of the project, F. van Dijk, directly from the results of Stage 2 and Stage 3, as published. The compiler is described in [VAN DIJK 82].

4.2.4. Evaluation

The structured approach as explained above has resulted in a design in which no significant errors or omissions have come to light.

It should, however, be pointed out that the very structuredness of the approach has turned compiler designing into a bookkeeper's job. Hardly any inspiration was needed since each step followed more or less mechanically from the previous one. Consequently, the design in chapter 5 is in essence a dull and detail-ridden work, in spite of or perhaps because of its obvious correctness.

Concluding paragraphs like 5.1.2.1.11 and 5.1.2.2.6 are symptomatic of the top-down approach and correspond to 'returns' from subroutine calls.

Now that this kind of work has been done by hand once, we are probably in a position to enlist mechanical aid if this design technique is repeated for another compiler. If I had to design another compiler (with an equally fitting and well-defined target code as ALICE is), I would let the computer keep track of the requirements. Each requirement would be labelled with what kind of information it is concerned with, who is interested, who is going to supply the information, and probably some other items. A program could then sort them so that no information would be needed before it was produced; clashes would be reported, requiring mending by hand. This process bears an interesting resemblance to the data-flow analysis often done by optimizing compilers [AHO & ULLMAN 78]. There the items tracked are run-time

values, here they are design-time values, i.e., the pieces of information needed for generating code.

Perhaps computer-aided compiler design is as feasible as, or even more feasible than, computer-aided compiler construction. For a system that seems to have all the necessary features, see [WILLIS 81].

4.3. The parser

A compiler traditionally consists of a sequence of N programs, each performing a pass over a representation of the source program and each producing tables and a transformed source program for its successor. Such a compiler is then called an N-pass compiler.

The present ALEPH compiler barely fits this description. It starts, as usual, by breaking up the sequence of characters which constitute the ALEPH text into units that correspond more or less to the symbols in ALEPH Manual 7.2. The parser then attempts to structure the resulting sequence according to a variant of the grammar of ALEPH (4.3.2). But rather than writing the augmented input to a file and passing it to a second program, the compiler distributes the information over a number of "information streams". Similar information goes to the same stream.

The ALEPH translator (= ALICE generator) then processes these information streams in the order required by ALICE (which differs completely from the order in which they were generated, see 4.2.2.1), and generates code from them.

Some aspects of the parser are treated below. For the details, especially concerning the error recovery, see [VAN DIJK 82].

4.3.1. The information streams

The information streams are implemented through ALEPH stacks and files. If the information written to a stream is needed again by the parser in some later stage, that stream has to be on a stack. If, however, the information is immaterial for further parsing, either a stack or a file can be used. Since one of the requirements in the compiler design was minimal memory usage, we use files wherever possible. Every piece of information that will not be needed again by the parser is immediately written to a file. Often information was split in a small part to be kept on a stack and to be consulted again, and a larger part to be written to a file and to be passed to the translator.

Such an aggregate of files is a very handy device for information sorting, both because of its ease of programming and because of its efficiency. It bears a close resemblance to a railroad switchyard where the vans from trains are regrouped into other trains according to their destination.

4.3.2. The input grammar

The ALEPH text is read according to an LL(1)-type grammar (given in [VAN DIJK 82]) which was derived by hand from the original ALEPH grammar in the ALEPH Manual. Many techniques for turning a grammar into an LL(1)-type variant are described in appendix C of [LEWIS II, ROSENKRANTZ & STEARNS 76].

Major surgery was necessary for three notions: **member**, **compound-member** and **expression**.

A **member** can start with a **tag** in the following ways (as indicated by the LL(1)-checking program from [GRUNE, MEERTENS & VAN VLIET 73]).

qwert + 3, ...	*$* **rule-tag** *in an* **affix-form**
qwert → yuiop, ...	*$* **source** *in a* **transport**
qwert = 3, ...	*$* **source** *in an* **identity**
qwert[yuiop] ...	*$* **list-tag** *in a* **source**
*qwert*yuiop[asdfg] ...*	*$* **selector** *in a* **source**

A **compound-member** can start in three ways with a **tag**:

(qwert: ...	*$* **rule-tag** *in a* **compound-member**
(qwert − yuiop: ...	*$* **rule-tag** *in a* **compound-member**
(qwert + yuiop, ...	*$* **rule-tag** *in an* **affix-form**

and in two ways with a **minus-unit**:

(− a: ...	*$* **local-part**
(−)	*$* **failure-symbol**.

The notion **expression** needed rewriting since it was left-recursive in the original version.

The LL(1)-property of the final version has been checked with the parser-generator PGEN written by G. Florijn and G. Rolf [FLORIJN & ROLF 81].

4.3.3. The derivation of the parser

The parser was derived from the LL(1)-grammar by human interaction with the original ALEPH compiler. As a first step all rules in the grammar were preceded by the symbol *PREDICATE*. The resulting ALEPH program was fed into the ALEPH compiler, which produced a number of error messages about rules that could not fail (since they contained an empty **alternative**) and warnings about backtrack. The rules that could not fail were made 'actions', which resulted in other error messages, now of two types: 'predicate cannot fail', remedied by turning the rule into an action, and 'alternative never reached', remedied by some rearranging. After a few turns only backtrack warnings remained. Each such warning points at a situation where a certain input *must* be present, but where the rule reading that input is a predicate. If the rule fails, an error message must be given and possibly some error correction must be done on the input stream and/or on the data structures. The offending rule call was therefore replaced by a **compound-member** consisting of that rule call as its first **alternative** and the error reporting and correcting actions as its second **alternative**. This done, we had in our hands an ALEPH syntax checker.

The next step was the insertion of actions that would derive output from the information just read and send it to the desired stream. Given the parallelism of these streams hardly any reordering of information was necessary.

A slight problem, however, arises where the grammar has been mutilated in order to make it of type LL(1). Here information is gathered and kept in affixes until a point is reached where the information can be written to the appropriate stream.

In a sense we are playing parser generator, but, as opposed to the average parser generator (as, e.g., Yacc [JOHNSON & LESK 78] or PGEN [FLORIJN & ROLF 81]), this approach allows us full control over the flow of information.

The result of these processes can be observed in the following excerpt from the parser.

```
PREDICATE member − tg:
    tag + tg, member after tag + tg;
    no tag member.

ACTION member after tag + >tg − src:
    source after tag + tg + src,
        (transport or identity tail + nil + src;
        error + no transport or identity tail + end of member,
            dummy member
        );
    transport or identity tail + tg + nil;
    actual affix sequence option + tg.

PREDICATE source after tag + >tg + src> − tg1:
    of unit,
        (tag + tg1;
        error + no tag + end of tag, tg → tg1),
        (non starred element + tg + tg1 + src;
        error + no subbus + end of source, dummy src → src);
    non starred element + tg + tg + src.
```

First the **tag** and then the **source** are kept in affixes rather than being written to a stream. The effects of the undeclared rules are given below. The phrase 'if possible' indicates that the predicate will fail if the described action is not possible.

```
PREDICATE tag + tg>:
    $ if possible, read a tag and yield a representative
    $ pointer in 'tg'.

PREDICATE no tag member:
    $ if possible, read a no-tag-member and write
    $ its translation.

PREDICATE transport or identity tail + >tg + >src:
    $ if possible, read a transport-or-identity-tail
    $ and write its translation. A pointer representing
    $ the (left-most) source is given in 'src'; if this
    $ is NIL, the source is a single tag represented
    $ by 'tg'.

ACTION error + >msg + >eon:
    $ display somewhere the error message represented by 'msg'
    $ and advance the input stream until a character is found
    $ that occurs in the set of characters indicated by 'eon'
    $ ('end of notion').
```

ACTION dummy member:
$ write the translation of a dummy **member**.

ACTION actual affix sequence + >tg:
$ write the translation of an **affix-form** *with a* **rule-tag**
$ represented by 'tg' and **actual-affix**es *still to be read.*

PREDICATE of unit:
$ if possible, read an **of-unit** *(a '*').*

PREDICATE non starred element + >tg0 + >tg1 + src>:
$ if possible, read a **non-starred-element** *(i.e., a*
$ **source** *between square brackets). Combine it with*
$ the **selector** *'tg0' and the* **list-tag** *'tg1' into*
$ a **source** *and yield a representative pointer in 'src'.*

4.4. On ALICE

ALICE (*AL*eph *I*ntermediate Cod*E*) was designed by A.P.W Böhm to serve as a machine-independent intermediate code; its original version is described in the ALICE Manual [BÖHM 77]. This chapter gives a short introduction, followed by some comments on the design. Then some problems are pointed out, and it is shown that the design technique had to be made more explicit to solve these problems.

Note: although the key-words in the ALICE Manual are written in lower-case letters, they are represented in capitals in this thesis to improve readability.

4.4.1. A short introduction to ALICE

An ALICE **program** results from the translation of an ALEPH **program** and consists of five sections:

- **status-information**: some general information about the ALEPH program, e.g., its name, the number of files it uses, etc.
- **values**: a list of identified constants used by the program; some are given explicitly, some must still be calculated.
- **data**: declarations of global variables, stacks, tables and files.
- **communication-area**: data for the interface with the run-time system.
- **rules**: the translation of the ALEPH **rule-declaration**s.

The textual appearance of an ALICE program is that of a bare assembler program; even the ALEPH identifiers have been replaced by integers (their **repr**s). The original identifiers are retained in special places for run-time error reporting.

An ALICE program is intended to be processed by a macro processor (or equivalent): each line contains one "instruction", consisting of a three-letter keyword followed by zero or more parameters. Some of these instructions carry macro-processor information only, but most are intended to cause code production on some machine (but may be ignored on others). In general each instruction contains all the information needed to generate the intended code. As a result some information is repeated many times in the ALICE program.

ALICE has 81 instructions, distributed as follows:

values	9
data	17
calling mechanism	18
affix-passing	18
classification	6
extension	4
miscellaneous	11
Total	83

As remarked before, many of these are redundant (on any given machine but not in general!). For instance, our Cyber implementation generates code for 40 of them.

Some of the instructions are used for calling 'standard external rule' like *plus*, *get char* or *pack string*. They carry a three-letter parameter identifying the external rule called; there are 74 of these.

The ALEPH **data-declaration**s map fairly directly on sequences of ALICE macros, except that all constants used in the program appear together in **values**.

The translation of an ALEPH **rule-body** is given as a directed graph, each node of which corresponds more or less to a **member** in the ALEPH text. This graph is linearized by giving each node a number (an 'address') and specifying the addresses of its success- and failure-nodes. The order of the nodes may differ completely from that of the corresponding **member**s.

A prominent feature of the ALICE **call** is the 'gate', a set of generalized registers which carry the parameters during the transfer from caller to callee. The flow of data to and from this gate is channelled through two registers, v_reg and w_reg, respectively. Another register, a_reg, is used to hold addresses of stacks, files, etc. Depending on the implementation technique chosen, these registers may correspond to real registers, may be incorporated in machine instructions, or may be dealt with otherwise.

We shall now show a typical node. We assume that the ALEPH program contains a **rule-declaration** whose heading is

QUESTION halve + >k + l>

(let us say that *halve* succeeds if *k* is even and then yields the half of *k* in *l*; otherwise it fails). A call

halve + p + q

where *p* is a global variable and *q* is a local (or formal) variable will then result in a node similar to the following. (Explanations have been added behind *$*s; this is not allowed in ALICE.)

```
LAB 27          $ This is node 27.
CLL 51,1,0      $ Call begins, 51 = repr of 'halve',
                $ 1 = can fail, 0 = is not recursive.
IGT 1           $ The parameter transfer area ('gate')
                $ has size 1.
LVV 72          $ Load V_reg with the value of global
                $ Variable 'p'; 72 = repr of 'p'.
CVR 1,1         $ Copy V_reg to either gate location 1
                $ or to stack location 1 ('k').
FCL 51,33       $ Fallible call of rule 51; on failure continue
                $ at node 33; on success continue here.
LDW 1,2         $ Load W_reg from either gate location 1
                $ or from stack location 2 ('l').
SWS 5           $ Store W_reg in local variable 'q';
                $ 5 = the stack location of 'q'.
FRE             $ Free W_reg
CLE 0           $ Call ends; 0 = continue at textually
                $ following node.
```

It is tempting to consider ALICE as the assembler language of an ALICE machine. This view, however, is artificial and misleading: ALICE macros have a meaning only in a very specific context, and the information in their parameters has a high degree of redundancy (e.g., an *FCL* may only occur after a *CLL* with the same first parameter). Neither of these aspects is found in a traditonal assembler language.

4.4.2. The design of ALICE

Aside from the obvious requirement that it should be able to mimic faithfully the semantics of ALEPH, ALICE was designed according to the following criteria (given in the order of decreasing priority):

- It should not add any machine-dependence.
  As to data, a direct consequence is that **integral-denotation**s, **character-denotation**s and **string-denotation**s should still possess their original forms. Alignments are no problem since the ALICE machine (as opposed to the ALICE language) has only one data item, integer.
  As to instructions, this means that we cannot make any assumptions on the nature of, e.g., the subroutine-jump.
- It must be possible to obtain reasonable code with reasonable effort on a variety of machines.
- The translation from ALEPH to ALICE should be reasonably straightforward.
  This was added to make our own lives easier and to prevent designs that would make the ALEPH compiler too slow.

The 'reasonable effort' required from the user to transform ALICE into acceptable code was interpreted as 'line-by-line macro processing'. More specifically we aimed at a structure in which each macro can be processed using only the information contained in its parameters.

The above criteria conflict (of course), but not very much so. The main clash is between instruction-independence and reasonably good code. If we want total

instruction-independence we are not allowed to make any assumptions about the internal structure of instructions and cannot supply any useful information; the code quality will suffer.

Two approaches are conceivable:

- Pass only essential information and rely on the receiver to find other information needed.
- Attempt to guess what the receiver will need on a variety of machines.

If we choose the first approach the receiver will probably not go through the trouble of performing a deep analysis and will produce second rate code. We have therefore chosen the second approach. It is not unreasonable to make certain assumptions about the properties of some machine-independent instructions. For instance, if asked: 'Will a subroutine call benefit from knowledge about the number of calls preceding it?', everybody will answer: 'No'. The certainty of the answer arises from assumptions about the properties of a (general) subroutine-call mechanism.

Moreover, the requirement of 'reasonable code through reasonable effort on a variety of machines' gives rise to some interesting concepts. Good examples in point are the **'repr-val-pair'** (ALICE Manual 3.2.1.1) and the 'gate' (ALICE Manual 3.3.2). A **repr-val-pair** is the ALICE form of an integer constant; it consists of two integers, viz., its representation and a reference to its value (not its value itself, since that may be unknown to the ALEPH compiler generating the ALICE **repr-val-pair**). The main operations on it are: **constant-source**, which declares a **repr-val-pair**, and **load-constant-in-v_reg**, which accesses it.

The assumed property underlying this concept is that on some machines constant values can be kept in machine-instructions, but not on all machines.

If the machine allows constants in instructions, the declaration can be ignored and the value is used directly at all times. Otherwise the **constant-source** macro results in a memory location labelled with the **repr** and filled with the value of the **valref**; access is then through the label. On the Cyber all constants in the range $-131071 : +131071$ get the first treatment; larger constants are kept in separate locations. Thus reasonable code is generated through reasonable effort on a variety of machines.

The technique used in the design of such concepts is the following: various scripts for the implementation of a feature are written down side by side and adjusted so that the actions in one script team up with comparable actions in the other scripts. These comparable actions may require different information, which is then supplied by various parameters.

The ALICE Manual shows the result of this process. A test implementation of ALICE was made on the PDP11/45 [BÖHM 78].

As explained above, the receiver of ALICE code will have to write an ALICE-to-object translator. In recent years the problem of the automatic generation of this type of translator has been taken up [CATTELL 80] as part of the PQCC project at Carnegie-Mellon University [LEVERETT et al. 80]. Here text in TCOL, a machine-independent intermediate code of a somewhat lower level than ALICE, is matched against a machine-description formalized in a TCOL-oriented way. The matchings found are used for code generation.

The flavour of ALICE, which is mainly flow-of-control oriented, is so different from that of TCOL, which is mainly expression-oriented, that a comparison is difficult. At first sight a translator from ALICE into TCOL seems possible but would probably feel

unnatural.

4.4.3. Problems with and modifications to ALICE

When ALICE was put to serious use in the implementation of the machine-independent ALEPH compiler, many small inconsistencies were uncovered and a few problems had to be cured, this in spite of careful checking. This shows again that no amount of human reading can replace a field test (nor can any amount of field testing replace human reading).

The small inconsistencies were mainly just plain bugs, which were easily corrected. There was, e.g., no way to translate an ALEPH dummy-affix (ALEPH Manual 3.4) (i.e., an output parameter whose value gets lost) into ALICE: the ALICE macro sequence **restore-from-output-gate** requires the value to be stored in at least one place.

The four more substantial problems were:

- The grammar of ALICE is not of type LL(1).
- The calling sequence conflicts with the design criteria (it cannot be derived from the source text 'in a reasonably straightforward way').
- The ALICE **extension** is inadequate.
- A new flow-of-control instruction had been requested, which would replace the caller by the callee (this 'swap' instruction was desired for writing finite-state parsers in ALEPH).

The solution of these problems required some redesign of ALICE, the details of which are given in chapter 6. This paragraph contains some observations on that design process.

To understand the LL(1) problem we have to realize that there are two ways to parse an ALICE text: either according to a regular grammar (which simply describes a sequence of distinguishable macros), or according to the context-free grammar given in the ALICE Manual. It is this last grammar that is not of type LL(1): some notions have two or more alternatives that can start with the same notion **N**.

The LL(1) problem is a good illustration of the idea that design is often more an art than a science. In spite of the rationalizations in paragraph 6.3, there is no hard scientific reason why the ALICE grammar should be of type LL(1). An ALICE program is just a sequence of macros, and, if it is a correct ALICE program, each macro is used in its proper context and is meaningful on its own accord. Since the writer of the ALICE processor is not supposed to check the correctness of the ALICE programs generated by the ALEPH compiler, the need for parsing according to the context-free grammar will never arise. Nevertheless, when the context-free ALICE grammar was fed to an LL(1)-checking program [GRUNE, MEERTENS & VAN VLIET 73, FLORIJN & ROLF 81], it pointed emphatically at the trouble spots: the calling sequence and the **extension**. In repairing these trouble spots the LL(1) requirement, which was based solely on aesthetical considerations, proved to be of great help.

The cause of this effect is not easily discerned. The problem with the calling sequence was that it required the knowledge of the number of locals of the rule to be called (in a **target-stack-frame** macro). This knowledge is only available after the **actual-rule** of the called rule has been fully analyzed, since additional locals may be generated by the translation process (6.4). This would mean that all **actual-rule**s had to be analyzed completely (and the results kept!) before code generation could start,

thus laying an unacceptable burden on the compiler. So a scheme had to be devised which avoided the necessity of knowing the number of locals at the call.

Now, this problem (one bad parameter in one macro) did not cause the LL(1) violation, nor did the LL(1) violation cause this problem. If nevertheless they have a relation, it must be through a common cause, which I surmise is the immaturity of the design of the calling sequence. This view is supported by the redesign of the **extension** sequence (6.5).

More generally I hypothesize that any area that has received less than average attention in any designed object, be it a program, a family budget or a city plan, has a larger chance of being implicated by any formal analysis, however unrelated, than the other areas. This may be the reason why all software checking tools (and all psychotherapy methods) help (a little).

If the cause works on more than one front, so does the cure. The wish to make the sequence both implementable and LL(1) serves to focus the attention, which in turn leads to a more mature, implementable, efficient and aesthetic design (as given in chapter 6).

The parallel-script design technique for machine-independent instructions as explained above can be seen at work in chapter 6 in the correcting of the calling sequence, in the design of the swap instruction and in the redesign of the **extension** sequence. In all cases the scripts catered for two different types of machines, those with registers (like the CDC Cyber where memory-to-memory operations are non-existent) and those without registers (like the PDP11/45, where, although it has registers, memory-to-memory operations are more efficient for the translation of ALEPH [BÖHM 78]). In addition to these two types, some thought was given to machines on which indirect addressing is to be avoided. It is remarkable to see that the resulting additional script greatly simplified the swap instruction.

We now have a good view of the development of the design technique itself. First the **repr-val-pair** more or less suggested itself in answer to our attempts to construct a machine-independent constant. Then the same happened in the design of the parameter passing, resulting in the concept of a 'gate'. We then realized that in both cases two different scripts were rolling off in parallel. This was then used as a point of departure in the correction of the calling sequence and as a life line in the redesign of the **extension** sequence, where both scripts had to be adjusted heavily to achieve a measure of flexibility which would not have been reached otherwise.

4.5. Bootstrapping

A verbal description of a bootstrapping process is notoriously long-winded. A better representation is that through T-diagrams [EARLEY & STURGIS 70]. As an experiment we shall introduce here a simple formalism for handling job steps and use it to describe the bootstrapping of the ALEPH compiler.

4.5.1. A formalism for job steps

We shall explain here a formalism which has some advantages over T-diagrams, although it is isomorphic to them.

In this formalism a program *P* in the language *LAN* which expects input in the language *INPUT* and yields output in the language *OUTPUT* is written as

$P = INPUT > OUTPUT\ ?\ LAN.$

Input in the language *INPUT* can be supplied to *P* by prefixing it with *INPUT* +, and running power on a machine capable of running *LAN* is supplied by postfixing it with *!LAN*. (If we have such running power, we say that *! LAN* is an 'available machine'). The result is then

$R = INPUT + INPUT > OUTPUT\ ?\ LAN\ !\ LAN.$

By applying the two reduction rules of the formalism:

$A + A >$ $\Rightarrow$ *empty*, and
$?\ M\ !\ M$ $\Rightarrow$ *empty* if *!M* is an available machine,

this reduces to:

$R = OUTPUT$

which is of course what we want. Every reduction of the type '$?\ M\ !\ M \Rightarrow empty$' corresponds to a run on an actual machine. (It should be noted that, in spite of their appearance, the >, +, ? and ! are not operators. They are, in fact, just separators, governing the reduction rules.)

As an example we shall now describe the normal compile-load-&-go sequence of a program in, say, ALEPH. We need input: *INPUT* +, three programs:

the user program: $UP = INPUT > OUTPUT\ ?\ ALEPH,$
the compiler: $CP = ALEPH > OBJECT\ ?\ BIN,$
the loader: $LD = OBJECT > BIN\ ?\ BIN,$ and
an available machine: $!BIN.$

The program is fed to the compiler which is then run on *! BIN*, yielding a load-module, *LM*:

$$\begin{aligned} LM &= UP + CP\ !\ BIN = \\ &= INPUT > OUTPUT\ ?\ ALEPH + ALEPH > OBJECT\ ?\ BIN\ !\ BIN = \\ &= INPUT > OUTPUT\ ?\ OBJECT, \end{aligned}$$

through application of the +>-rule. We shall now do the load-&-go phase in one step, supplying the data in *INPUT*:

$RESULT = INPUT + LM + LD\ !\ BIN\ !\ BIN,$

thus calling for two machine-runs:

$$\begin{aligned} &RESULT \\ &\quad = INPUT + INPUT > OUTPUT\ ?\ OBJECT + \\ &\qquad\quad OBJECT > BIN\ ?\ BIN\ !\ BIN\ !\ BIN = \\ &\quad = OUTPUT\ ?\ BIN\ !\ BIN = OUTPUT. \end{aligned}$$

We need not do the reductions in this order and can, for instance, derive a formula for the general compile-load-&-go sequence in the absence of the user program and the input:

CLG
= *CP ! BIN + LD ! BIN ! BIN =*
= *ALEPH > OBJECT ? BIN ! BIN +*
OBJECT > BIN ? BIN ! BIN ! BIN =
= *ALEPH > BIN ! BIN.*

And indeed, if we prefix this with a user program and input it reduces to the desired output. However, since it is not of the form $A > B ? C$ it is not a program.

The above example shows that *+ ALEPH > BIN ! BIN* (where *! BIN* is an available machine), behaves as if it were *! ALEPH*, i.e., it is (almost) an available machine. This is generally true:

If there exists a program *S = M1 > M2 ? M3*, and *! M2* and *! M3* are available machines, then *! M1* is also an available machine, and *! M1 = + S ! M3 ! M2 = + M1 > M2 ! M2.*

Proof: *! M1* is an available machine if the reduction '*?M1!M1* $\Rightarrow$ *empty*' is allowed:

? M1 ! M1 = ? M1 + M1 > M2 ! M2 = ? M2 ! M2 = empty

since *! M2* is an available machine. (*M3* does not occur, since it is only used to drive the translator from *M1* to *M2*.)

Some advantages of this notation over the traditional T-diagrams are that it is easier to type, that reductions can be done conveniently even on incomplete jobs, and that it never gets geometrically stuck.

4.5.2. Bootstrapping the compiler

The proposed transporting scheme is now as follows (ALICE Manual 1.2).
The ALEPH compiler is brought to the target site, both in ALEPH and in ALICE:

P = ALEPH > ALICE ? ALEPH,
Q = ALEPH > ALICE ? ALICE.

An important property of ALICE is that it is a one-statement-a-line language, with a format which is easily accepted by most macro-processors, including those normally incorporated in assemblers. Moreover, the communication between the statements is very restricted, consisting mainly of a constant table; all other pertinent information is repeated in each statement. This makes it easy to translate each statement into some assembler instructions, independent of the other statements.

The receiver now writes (probably by hand) a macro-definition file *R* which converts ALICE to the target assembler, say, *TASS*,

R = ALICE > TASS ? MAC,

and runs

Q + R ! MAC = ALEPH > ALICE ? TASS = S.

He then constructs the job

T = S ! TASS + R ! MAC = ALEPH > TASS,

which produces *TASS*, and with which he can run an ALEPH program *K = INPUT > OUTPUT ? ALEPH*:

INPUT + K + T ! TASS = OUTPUT,

at the expense of three runs. (Actually, many more runs will be necessary, since the operations *! MAC* and *! TASS* will each be composed of a number of runs on the actual machine.) Corrections and improvements found during this debugging and learning phase can easily be effected by editing *R*, which is the only variable part.

After a while the situation stabilizes, and it becomes desirable to remove the *! MAC* step from the job step *T*. The obvious (but not the best) way is to write an ALICE processor *U*:

U = ALICE > TASS ? XYZ

in the most appropriate vernacular *XYZ* and to obtain binary code from it:

U' = ALICE > TASS ? BIN.

Now the program *K* can be run as follows:

DATA + K + S ! TASS + U' ! BIN ! TASS = OUTPUT,

which still takes three runs, but supposedly *! BIN* is much more efficient than *! MAC*.

A larger improvement can, however, be obtained by starting from the original ALEPH compiler *P* (which has not yet played a role). This program is structured so that the ALICE-generating part is easily isolated and replaced by a *TASS*-generating part. The structuring is based on the distinction between ALICE as a stream of (internal) information and ALICE as a stream of (external) characters.

The internal stream is represented in *P* as a sequence of calls of the rule *g macro*, one for each ALICE macro. At the moment of the call the pertinent parameters are available on the stack *pars*; the first parameter is an indication which macro is to be produced.

The supplied ALEPH compiler chooses this indication to be the address of a string describing the format of the macro and the nature of its parameters. All *g macro* has to do is to copy the string and to replace certain characters in it by certain parameters from *pars*.

The receiver can, however, replace *g macro* and a number of constants, and have the ALEPH-compiler *P* generate *TASS* (for details see [VAN DIJK 82]):

P' = ALEPH > TASS ? ALEPH.

This leads in an obvious way to the required form of the compiler:

P' + T = ALEPH > TASS ? TASS.

All the above hinges on the ease of implementation of ALICE, even when we have proceeded to a stage where no ALICE is explicitly produced any more. The underlying machine is still an ALICE machine, executing the ALICE primitives. Most of these are trivial, but two of them, extending stacks and input/output, need considerable attention.

Extending a stack is by itself a simple operation, but trouble arises when there is no more space. The simplest option is to give up, and this may be acceptable during the installation phase, but for production purposes it will soon be necessary to create room. Several schemes are given in ALICE Manual 3.2.2.1. As explained in hint 6, a

stack-shifting algorithm is provided in ALEPH (and in ALICE), to aid in implementing the **extension** primitive.

Input/output is as complicated as the operating system requires. Very little machine-independent support can be given here.

5. THE DESIGN OF THE ALEPH COMPILER

As explained in 4.2.3 the ALEPH compiler was designed in four stages:

- stage 1, find ALICE translations of all ALEPH constructs,
- stage 2, take stock of the information items needed by each ALICE construct,
- stage 3, devise ways to obtain and process this information,
- stage 4, design actual algorithms and concrete data representations.

Stages 1 & 2 were combined into a stock-taking phase; stage 4, the concretization phase, was for the larger part incorporated in the actual writing of the compiler [VAN DIJK 82].

The results of the design technique are shown below, stages 1 & 2 in 5.1 and stage 3 in 5.2. To prevent the reader from being suffocated by details, the reporting has been restricted to the first two sections of the ALICE code, **status-information** and **values** (see 4.4.1). The design process is depicted in Fig 12, where the design tasks are performed in left-to-right top-to-bottom order; the shading indicates the tasks described in this thesis.

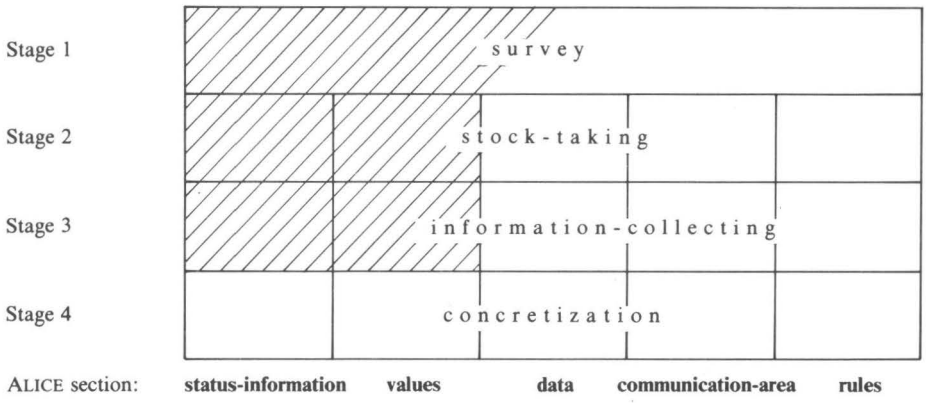

Fig. 12.

It will be clear that such a transversal cut through an iterative design cannot be made without impunity. Problems arising from tasks being left undescribed were solved by referring to the pertinent parts of the ALEPH and ALICE Manuals.

The stages 1, 2 and 3 involve some abstract algorithms in which (compile-time) variables occur. If the value of such a variable V is used as part of the name of some other entity, it is written [V]. So, if the variable *last stack* has been set to the name of the stack *profit*, then '<<[*last stack*]' means '<<*profit*'. The precise ways in which these items are represented in the actual compiler are decided on in stage 4.

5.1. The tasks of the compiler

The first goal of the ALEPH compiler is to provide the user with:

- a translation of his ALEPH program into ALICE,
- a listing,
- possibly a cross-reference,
- syntactic-error messages where appropriate,
- semantic-error messages where appropriate.

Second to that, we want the compiler to run, possibly slowly, on a small machine and to be easily adaptable to a bigger one (4.2.1). This obliges us to keep direct-access data to a minimum, an obligation which will profoundly affect our design.

As explained in the ALICE Manual and 6.7 the ALICE program resulting from a call of the ALEPH compiler consists of five sections:

status-information,
values,
data,
communication-area, and
rules.

These sections must be constructed from information gathered by the compiler. So in very broad outline the compiler can be described as:

ACTION compile program:
create status information &
create values &
create data &
create communication area &
create rules.

The semantics of the ampersand (*&*) will have to remain vague. The intention is that the various components of the five "processes" are executed in such an order as to yield correct results. The ampersand does not imply that its left side and right side are executed collaterally in the sense of ALGOL 68, but only that we have not yet decided about their synchronization. The final design will, of course, not contain any ampersands.

In the meantime this feature enables us to talk about *create-status-information* as a process in its own right, rather than a set of actions spread out over the whole compiler.

We shall now turn to the five sections of the ALICE program.

5.1.1. *Create-status-information*

To produce the ALICE code for **status-information** (ALICE Manual 3.4) we must have the following information:

a) the title **string** of the **program**,
b) the maximum of all **size-of-input-gate**s and **size-of-output-gate**s (ALICE Manual 3.3.2),

c) the number of **value**s,
d) the number of **variable-decl**s,
e) the number of **file-administration**s,
f) the number of breathing lists (ALICE Manual 3.2.2.1),
g) the number of non-breathing lists,
h) background option,
i) dump option.

Since most of these items can be defined or modified almost anywhere in the program, it is clear that we must read the entire program text before we can generate the first ALICE instructions. All this information and much more must be gathered in data structures to be produced at command.

Items a, h and i result from **pragmat**s or the absence thereof.

Items d, e, f and g can be determined by simple counting.

Since **size-of-input-gate** and **size-of-output-gate** (item b) for a rule follow directly from its heading, their maximum can easily be established.

Item c, the number of (ALICE) **value**s, however, is the result of a thorough transformation of the **constant-declaration**s in the ALEPH program. This implies that the transformation algorithm must have a way to tell in advance how many **value**s it will generate. See 5.1.2.2.2.

5.1.2. *Create-values*

The ALICE-part **values** (ALICE Manual 3.1) consists of the collection of all values, integer, character, pointer, etc., that are used in the rest of the ALICE program, i.e., in the ALICE **data**, **communication-area** and **rules**. Expressions defining these values are submitted to the ALICE processor which is the first program to be able to evaluate them. The resulting values are assigned unique representations (called "**valref**s") and are referred to in **data** and **rules**, in which no other values occur. The expressions and values in **values** are partially ordered in such a way that no value is ever referenced until after its initialization. This order need not be the same as in the ALEPH program:

CONSTANT p = q − 1.
CONSTANT q = 15.

Here *q* is referenced (textually) before being initialized; the semantics of ALEPH Manual 3.1.1 makes this legal.

We shall have to do some sorting, which can only be done after all expressions and values have been met. An additional result of the sorting must be the number of values to be generated (5.1.1). This gives us the following structure for *create-values*:

ACTION create values:
    collect values, sort and count and output values.

The valrefs in ALICE are represented by integers in such a way that they appear in the ALICE-values in a contiguous ascending sequence. Since the order of the **value**s is determined by sorting, it is clear that *collect-values* cannot assign the correct valrefs to the values it finds. We shall therefore let *collect-values* generate provisional valrefs, called "defrefs", the nature of which will be determined in the process of defining *collect-values*.

5.1.2.1. *Collect-values*

The process *collect-values* must identify all constructions in the program that give rise to (compile-time) values and assign defrefs to them. When looking through the ALEPH Manual we find the following items from which all other values derive:

integral-denotations,
character-denotations,
constant-tags, and
table-**limit**s.

Note that these are in fact the members of **plain-value**, which can be used in **base**s, **term**s and **expression**s to form new constant values.

Now for each value in the program we must provide a defref and enough information for that value to be calculated.

5.1.2.1.1. Plain-values

We shall first consider the four alternatives of **plain-value**.

Integral- and **character-denotation**s are no problem: instructions for assigning valrefs to them exist in ALICE. An intermediate defref will not do any harm.

Constant-tags have already got a representation, which can act as a defref; they obtain their values in **constant-description**s or in **pointer-initialization**s.

The **tag**s in **external-constant-description**s do not give rise to compile-time constants and need not be considered here.

A **constant-description** equates a **constant-tag** to an **expression**, which we shall deal with later on (5.1.2.1.5).

On the other hand, the **pointer-initialization** is a problem. It gives the value of the **constant-tag** as the virtual address of the preceding block in a **filling-list-pack**. This address is dependent on the way virtual memory is allocated, as described in ALEPH Manual 4.1.4. The recipe presented there supplies the min-limit of the stack in which the **pointer-initialization** occurs, provided we know the lengths of all list **filling**s of all tables and stacks without **size-estimate** and the values of all **expression**s in **absolute-size**s and in **relative-size**s. We can then calculate the desired value from the value of the min-limit and the offset of the block from the beginning of the **filling**. If that's what it takes, that's what it takes.

Table-limits exist in three forms, **min-limit**s, **max-limit**s and **calibre**s. Their representations (of the form $<<TAG$, $>>TAG$ and $<>TAG$) can be used as defrefs. The value of a **calibre** is the number of **selector**s in the **stack-** or **table-head**, so it is known to the compiler in a machine-independent way. The values of **min-** and **max-limit**s are provided by the mechanism loosely described above.

The conclusion is that, if we are able to evaluate **expression**s and do all the calculations indicated in ALEPH Manual 4.1.4, we can indeed supply ALICE **value**s for all the members of **plain-value**. We shall leave the details of this process until after the treatment of the following problem.

5.1.2.1.2. An inventory of values

All values originate in **expression**s or in 'constant-sources', where "constant-source" is that part of **source** which could also occur as a **plain-value**. Since ALICE requires a complete list of values in its **values**, *collect-values* will have to recognize all **expression**s and constant-sources. This causes some problems.

It may be noted that **character-denotation**s, **constant-tag**s and table-**limit**s occur exclusively in **plain-value**s. **Integral-denotation**s occur in **plain-value**s and in **pragmat-item**s. In the latter case they appear in ALICE as **string**s rather than as values and need not be considered here.

5.1.2.1.2.1. Recognizing expressions

Expressions occur in:

exits,
zones,
expressions (recursively),
constant-descriptions,
variable-descriptions,
single-blocks,
compound-blocks,
relative-sizes, and
absolute-sizes.

Each of these can be recognized without problems, except for the **expression** in a **zone**, where it clashes with a single **list-tag**. If we find a single **tag** in a **zone**, we have a problem. If it is a **constant-tag**, it is an **expression** for which normal expression code must be generated, and if it is a **list-tag** it must be left in place. We shall see, however, that the code generated for an **expression** which consists of a single **tag** is that same **tag**, so that in practice the problem does not occur (5.1.2.1.9).

5.1.2.1.2.2. Recognizing constant-sources

Constant-sources occur in **source**s where they appear alongside

table-**element**s,
variable-tags,
stack-**limit**s,
stack-**element**s, and
dummy-symbols.

If the **source** happens to occur as an **actual**, further side-lines appear:

list-tags and
file-tags.

Until we have read the whole ALEPH program, we cannot with certainty distinguish **variable-tag**s, **list-tag**s and **file-tag**s from **constant-tag**s, nor stack-**limit**s from table-**limit**s. This means that we shall have to collect information about the use in a provisional form first and combine it later with declaration information.

5.1.2.1.3. Definitions as generated by *collect-values*

We are now in a position to give a complete list of all constructs to be examined by *collect-values* and to state what is to be done in each case.

At this level of description *collect-values* will yield a sequence of 'definitions', which we shall write down in a form similar to a **constant-description**:

tag, equals symbol, expression.

Since we shall need more tags than are present in the program, we shall allow tags to contain the special characters <, >, *!* and #. We shall use these tags as defrefs.

5.1.2.1.4. Hidden definitions

Some of the examined constructs contain **expressions**, and others give rise to implicit definitions with hidden expressions, as we saw from the above **pointer-initializations**. We shall first make all implicit definitions explicit, so that the problem reduces to the treatment of straightforward definitions.

Implicit definitions exist in **table-heads** (ALEPH Manual 4.1.5), **stack-heads** (ALEPH Manual 4.1.6), **filling-list-packs** and **pointer-initializations** (both ALEPH Manual 4.1.5). Each **table-** and **stack-head** of a list *LST* is an implicit definition of the calibre *<>LST* and the min-limit *<<LST*. The max-limit *>>LST* derives from the length of the **filling-list-pack**. Two more values figure in the explanation in ALEPH Manual 4.1.4, the 'virtual-min-limit', here written as *!<LST*, and the 'virtual-max-limit', *>!LST*.

The meanings of these values are shown in Figure 13, where a stack with calibre 3 and containing 5 blocks is displayed.

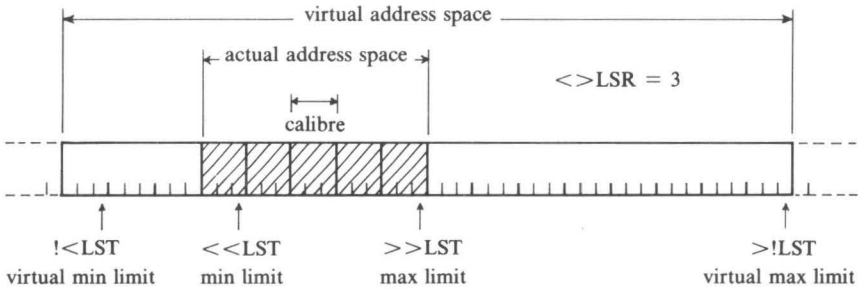

Fig. 13.

The highest address that ever can occur in a calculation is the virtual-max-limit of the right-most list. The lowest possible address is the address just left of the left-most

list; this address must be available, i.e., its calculation may not cause (negative) integer overflow (although the corresponding location need not exist).

Not all of these values may occur in **expressions**, but they do contain all the information that may ever be asked about a stack or a table. If some of them should turn out to be superfluous, they could be omitted afterwards.

5.1.2.1.4.1. Hidden definitions from list-heads

The recipe in ALEPH Manual 4.1.4 distinguishes between tables and stacks without **size-estimate** (called here 'fixed lists'), stacks with an **absolute-size** and stacks with a **relative-size**; so shall we.

5.1.2.1.4.1.1. Definitions generated for fixed-lists

The name of the latest fixed-list is kept in the variable *last fixed* which is initially set to $\#FL$, the name of the (virtual) fixed-list before all fixed-lists.

Each head with the tag FL and a calibre CAL yields the following definitions:

$$!{<}FL = {>}![\textit{last fixed}] + {<>}FL$$
$${<>}FL = CAL$$
$${>}!FL = {>>}FL$$

and *last fixed* is set to FL. We do not need to generate a definition for $<<FL$, since it is equal to $!<FL$ when the program starts (ALEPH Manual 4.1.4) (although the two values may diverge later on, due to calls of *unqueue* or *unqueue n*).

In the case of an external-table with string STR the last definition is replaced by:

$${>}!FL = {>}![\textit{last fixed}] + \textit{external table size}(STR)$$

Note that the value of $>>FL$ cannot be deduced from the list-head. It will be defined in 5.1.2.1.4.2, where it originates from the **filling-list-pack**. Since definitions from the program may be out of order anyway we need not have compunctions about generating this one out of place.

5.1.2.1.4.1.2. Definitions generated for absolute-size stacks

The name of the latest absolute-size stack is kept in a variable *last ast* which is initially set to $\#AST$.

Each head with tag AST, calibre CAL and **absolute-size** SIZ yields the following definitions:

$$!{<}AST = {>}![\textit{last ast}] + {<>}AST$$
$${<>}AST = CAL$$
$${>}!AST = {>}![\textit{last ast}] + SIZ$$

and *last ast* is set to AST.

5.1.2.1.4.1.3. Definitions generated for relative-size stacks

The name of the latest relative-size stack is kept in a variable *last rst* which is initially set to $\#RST$.

Each head with tag RST, calibre CAL and **relative-size** SIZ yields the following definitions:

!<RST = >![last rst] + <>RST
<>RST = CAL
>!RST = >![last rst] + virtsize!RST

sumsize!RST = sumsize![last rst] + SIZ
virtsize!RST = (virtleftover![last rst] / sizeleftover![last rst]) × SIZ
virtleftover!RST = virtleftover![last rst] − virtsize!RST
sizeleftover!RST = sizeleftover![last rst] − SIZ

and *last rst* is set to *RST*.

(Note the building of new defref names with special characters: for each stack *XXX* there are defref names like *sumsize!XXX*, etc.)

The last four definitions implement the proportional distribution required in ALEPH Manual 4.1.4.d. The order of division and multiplication has been chosen so as to avoid integer overflow. The four constants defined above have the following meanings:

sumsize!RST — the sum of all **relative-size**s of all relative-size stacks up to and including the stack with the tag *RST*,

sizeleftover!RST — the sum of all **relative-size**s of all relative-size stacks following the stack with the tag *RST*,

virtsize!RST — the size of the virtual memory allotted to the stack with the tag *RST*, and

virtleftover!RST — the sizes of the virtual memory allotted to all relative-size stacks following the stack with the tag *RST*.

When all list-heads have been processed, the following six constants will still be undefined:

>!#FL — the right-most address of the 'zero-th' fixed-list, i.e., the one address just before all lists, mentioned in 5.1.2.1.4,

>!#AST — the right-most address of the zero-th absolute-size stack, which is the last fixed-list, if it exists, or *>!#FL* otherwise,

>!#RST — the right-most address of the zero-th relative-size stack, which is the last absolute-size stack, if it exists, or *>!#AST* otherwise,

sumsize!#RST — the sum of the relative sizes of all relative-size stacks before the first, i.e., 0,

sizeleftover!#RST — the sum of the relative sizes of all relative-size stacks after the zero-th, if any, or 0 otherwise,

virtleftover!#RST — the amount of virtual memory available for all relative-size stacks.

This leads to the following definitions to be added at the end of the program (again happily out of order):

>!#FL = manifest constant(MNA)
>!#AST = >![last fixed]
>!#RST = >![last ast]
sumsize!#RST = 0
sizeleftover!#RST = sumsize![last rst]
virtleftover!#RST = manifest constant(MXA) − >![last ast]

Here *MNA* and *MXA* are the ALICE symbols for the (implementation-dependent) bounds of the virtual memory.

This scheme also works if some or all of the types of lists do not occur in the program.

If, however, the **expressions** for *SIZ* (the **absolute-** or **relative-sizes** of ALEPH stacks) evaluate to crazy values, strange things happen. A negative value for *SIZ* will result in a negative address space; if all *SIZ*es are zero, division by zero results. We have no way of safeguarding against this: the ALICE processor should be prepared to deal with such cases, as it will have to deal with a virtual address space that turns out to be smaller than the actual address space.

5.1.2.1.4.2. Hidden definitions from filling-list-packs

The definitions given so far fail to define the max-limit, which stands to reason since the latter cannot be deduced from the list-head but must be taken from the **filling-list-pack** instead. In ALEPH the **filling-list-pack** may be missing, but to simplify the discussion we shall assume the presence of a **filling-list-pack** for each list-definition; if need be an empty (and in ALEPH illegal) **filling-list-pack** '*= ( )*' can be assumed.

The processing of a **filling-list-pack** in the definition of a list *LST* with calibre *CAL* requires three variables: a variable *last pointer* which is initialized to *>![prev lst]*, where *prev lst* is a global variable referring to the name of the previous list of the same type as the present one; a variable *offset* which is initially set to 0; and a counter *n* starting at 1.

For each **single-** or **compound-block** (which must be of length *CAL*), *offset* is increased by *CAL*, and no definition is generated.

For each **string-denotation** of length *K* the following definition is generated:

#[n]LST = [last pointer] + offset + stringlength(K)

and

last pointer is set to *#[n]LST*,
offset is set to 0 and
n is increased by 1.

For each **pointer-initialization** with **tag** *PNT* we generate:

PNT = [last pointer] + offset

and set *last pointer* to *PNT* and *offset* to 0.

At the end of a **filling-list-pack** we generate:

>>LST = [last pointer] + offset .

The ALICE **list-area** (ALICE Manual 3.2.2.1) requires a valref for the number of virtual addresses. For a fixed-list with tag *FL* this is '*>>FL − >![last fixed]*', for an external-table it is *external table size(STR)*, for an absolute-size stack it is its **absolute-size** *SIZ*, and for a relative-size stack with tag *RST* it is *virtsize!RST*. Since each **table-** or **stack-head** is followed by a **filling-list-pack**, one variable *virt size* suffices.

This concludes the treatment of list-heads and **pointer-initialization**s.

5.1.2.1.5. Definitions from constant-descriptions

For each **constant-description** with **tag** *TAG* and **expression** *EXP* we generate the definition:

TAG = EXP

5.1.2.1.6. Definitions from naming unnamed values

We have now covered all named values. Unnamed values are **integral-** and **character-denotation**s in constant-sources and **expression**s.

The difference between named and unnamed values is important because ALICE supports arithmetic only if it is dyadic on simple named values; and if a constant value appears as a source in ALICE, it must be a named value.

We shall therefore name all values and generate 'secret' defrefs as required. For this we need a global variable *defref count*, starting at 1.

For each **integral-denotation** *INT* not in a **pragmat-item** we generate:

#[defref count] = int denotation (INT)

and increase *defref count* by 1.

For each **character-denotation** *CH* we generate:

#[defref count] = char denotation (CH)

and increase *defref count* by 1.

For each **expression** *EXP* in an **exit**, a **zone**, an **expression**, a **variable-description** or a **single-** or **compound-block**, we generate:

#[defref count] = EXP

and increase *defref count* by 1.

The same is done for each of the components of **base**, **term** and **expression**. This yields definitions of the following form:

base defref = *plain value defref*
base defref = *(expression defref)*
term defref = *base defref*
term defref = *term defref* × *base defref*
term defref = *term defref / base defref*
expression defref = *term defref*
expression defref = + *term defref*
expression defref = − *term defref*
expression defref = *expression defref* + *term defref*
expression defref = *expression defref* − *term defref*

At this point the story may become boring, but at least it is complete to the point of exhaustion.

Two expressions which resulted from hidden definitions have unnamed values in them, for which defrefs can be created in the same way. Said expressions occur in the definitions of *virtsize!RST* in paragraph 5.1.2.1.4.1.3, and of *#[n]LST* in 5.1.2.1.4.2.

We have now assigned defrefs to and generated definitions for all constant values in the program.

5.1.2.1.7. The place of *collect-values* in the total scheme

We can visualize the function of *collect-values* as follows. The process *collect-values* reads the text of the ALEPH program and produces two texts: a list of definitions of **constant-tags** (defrefs), and a copy of the program from which all **constant-descriptions** have been deleted and in which each constant-source is replaced by a **constant-tag**. If we changed the format of the list of definitions into that of a large **constant-declaration** and concatenated both texts, we would obtain a new program that is semantically identical to the original program (if we accept the explicit calculation of virtual addresses which cannot be specified in official ALEPH).

This is a step in the right direction since the list of definitions can serve as a basis for generating the **values** part of the ALICE text, and the copy of the program and the data- and rules-part of the ALICE code are similar in that both contain the same information and neither contains unnamed values.

5.1.2.1.8. An example

Suppose the source code contains the **zone** *[−3×(/A/ − /a/)]*. This results in the following 15 definitions:

#1	=	3	$ plain value	
#2	=	#1	$ base	*
#3	=	#2	$ term	*
#4	=	/A/	$ plain value	
#5	=	#4	$ base	*
#6	=	#5	$ term	*
#7	=	#6	$ expression	*
#8	=	/a/	$ plain value	
#9	=	#8	$ base	*
#10	=	#9	$ term	*
#11	=	#7 − #10	$ expression	

```
#12  =  ( #11 )        $ expression pack   *
#13  =  #12             $ base              *
#14  =  #3 × #13        $ term
#15  =  − #14           $ expression        *
```

and the source code is copied as *#15.*

ALICE has no identity-operator, no monadic operators and no bracketing. So many of the above definitions cannot be expressed directly in ALICE; these are marked with an * in the last column.

5.1.2.1.9. The non-ALICE constructs

The question arises who is going to do something about this. At first sight it seems quite feasible to have *collect-values* contract all identities, monadic pluses and **expression-packs**, and add zeros to all monadic minuses. It should then deal automatically with cases like:

CONSTANT dog = cat, cat = (+mouse), mouse = /q/ − /s/.

and replace all *dogs*, *cats* and *mice* with the generated defref for */q/ − /s/*. In order to do this, however, it requires direct access to the list of definitions and to the copied program texts in which the animals occur. Now, although direct access to all definitions might be granted under protest, direct access to the program text is out of the question (4.2.1).

The list of definitions is inherited by the process *sort-values*, which will have to solve these problems anyway.

Nevertheless, if we wish, some things can be done by *collect-values* to simplify the produced list. We can generate

defref1 = defref2

for

defref1 = + defref2

and for

defref1 = (defref2),

and

defref1 = #0 − defref2

for

defref1 = − defref2,

if we start by issuing a definition

#0 = 0.

Moreover, any time a secret defref is about to be generated equal to an existing defref, generation can be omitted and the existing defref be used instead.

With this simplification made, the definition list for the **zone** above reduces to:

```
#1  =  3        $ plain value
#2  =  /A/      $ plain value
```

```
#3  =  /a/          $ plain value
#4  =  #2 - #3      $ expression
#5  =  #1 × #4      $ term
#6  =  #0 - #5      $ expression
```

This is a considerable reduction, well worth the effort, although it does not solve the general '*dog*, *cat* and *mouse*' problem above. But it does ensure that the code generated for an **expression** which is a single **tag** is that same **tag**, thereby fulfilling the promise of paragraph 5.1.2.1.2.1.

Another simplification may be obtained by observing that the *offset* in the definitions of *#[n]LST*, *PNT* and *>>LST* in paragraph 5.1.2.1.4.2 is often 0, namely after every **string-denotation** and **pointer-initialization**. But this modification does not affect the form of the definition list and a decision about it can be taken at any time (5.2.2.1.7).

5.1.2.1.10. The grammar of the definition list

The following forms occur in the definition list:

defref = *defref* $ *
defref = *defref* { +, −, ×, / } *defref*
defref = *int denotation (digit string)*
defref = *char denotation (char)*
defref = *string length (integer)*
defref = *manifest constant (symbol)*
defref = *external table size (string)*

The first one does not correspond to an ALICE construct; *sort-values* will have to take care of this.

The defrefs in the definition list may have the following forms:

TAG
!<TAG, *<<TAG*, *<>TAG*, *>>TAG*, *>!TAG*
sumsize!TAG
virtsize!TAG
virtleftover!TAG
sizeleftover!TAG
#[N]TAG
#[N]

where

TAG is a tag occurring in the program or
#FL, *#AST* or *#RST*

and

N is a compile-time integer variable

5.1.2.1.11. Conclusion

This concludes the stock-taking phase of the design of *collect-values*.

5.1.2.2. *Sort-and-count-and-output-values*

The list of definitions as obtained from *collect-values* is at least three steps away from the final goal, a sorted list of ALICE-values. The definitions are not sorted, they contain defrefs rather than valrefs, and one of them is not ALICE. On top of that, the definitions as extracted from the program may turn out to be circular, or involve undefined or incorrectly defined defrefs.

Sorting will require direct access: we are not going to do a polyphase sort-merge. Now that we have collected all definitions, and no longer have to worry about the program itself, we can afford to read them in in toto. With this direct-access facility the structure of *sort-and-count-and-output-values* becomes clearer:

ACTION sort and count and output values:
read values into direct access,
check and construct and output values,
discard values from direct access.

We must remember here the requirement from 5.1.1, that no ALICE **value**s may be output until their number is known and output in **status-information**. We shall delegate this to *read-values-into-direct-access* (5.1.2.2.2).

5.1.2.2.1. *Check-and-construct-and-output-values*

This process has five tasks:

- check for circularities and undefined defrefs,
- remove non-ALICE operations,
- sort and assign valrefs,
- yield a translation table of defrefs versus valrefs,
- output ALICE **value**s.

These activities are best combined in one algorithm consisting of three parts:

- a driver which makes sure that all definitions are handled,
- a definition processor which turns correct definitions into ALICE **value**s and
- a searcher which obtains a valref for a given defref.

The algorithm produces a stream of ALICE **value**-macros together with a translation table whose elements have the form *(defref, valref)*. It gradually deletes the entire definition list.

5.1.2.2.1.1. The driver

1) as long as there is a definition in the list, process that definition.

5.1.2.2.1.2. Processing a definition *D*

1) mark the definition *D* as UNDER CONSIDERATION (to catch circularities).
2) if *D* is of the form *defref1* = *defref2*, obtain a valref *v1* for *defref2*.
3) if the right-hand-side of *D* does not depend on defrefs, process it as follows.

3.1) if *D* is of the form

defref1 = int denotation (DIG),

obtain a new valref *v1* and generate

INT v1,DIG

3.2) similar actions for *char denotation (CH).*
3.3) similar actions for *string length (INT).*
3.4) similar actions for *manifest constant (SYM).*
3.5) similar actions for *external table size (STR).*

4) if the right-hand-side of *D* depends on defrefs, process it as follows.
4.1) if *D* is of the form

defref1 = defref2 + defref3,

obtain valrefs *v2* and *v3* for *defref2* and *defref3* respectively, obtain a new valref *v1* and generate

ADD v1,v2,v3

4.2) similar actions for

defref1 = defref2 − defref3.

4.3) similar actions for

defref1 = defref2 × defref3.

4.4) similar actions for

defref1 = defref2 / defref3.

5) enter the pair (*defref1*, *v1*) into the translation table.
6) remove the definition *D* from the list.
7) yield the valref *v1*.

5.1.2.2.1.3. Obtaining a valref *V* for a defref *DR*

1) if *DR* occurs in the translation table, yield the corresponding valref;
2) if no definition of *DR* occurs in the definition list, *DR* is an undeclared tag from the program; give an error message with *DR* (and line number) and yield the valref of zero;
3) if the definition of *DR* is marked UNDER CONSIDERATION a circularity exists; give an error-message with the last program tag and present line number and yield the valref of zero;
4) otherwise process the definition of *DR* and yield the valref thus obtained.

5.1.2.2.2. *Read-values-into-direct-access*

Although this operation seems trivial, there is one task it can fulfil. We need to know how many ALICE values there will be before generating the first one (5.1.1). Now, from the above algorithm we see that for each definition *D* there will be an ALICE value, except if *D* is of the form

defref = defref .

Definitions can be counted and distinguished by *read-values-into-direct-access*, and the

resulting number passed to *create-status-information*.

5.1.2.2.3. *Discard-values-from-direct-access*

The definitions can be discarded but the translation table must be kept.

5.1.2.2.4. Correctness

The following facts can be observed.

- All definitions are processed (because of the driver).
- Each definition generates one ALICE macro before it is removed (5.1.2.2.1.2.3 and 5.1.2.2.1.2.4), except when it is an identity (5.1.2.2.1.2.2). The reading process can, through simple counting, determine the number of ALICE macros to be produced.
- An ALICE macro with valrefs as second and/or third operands is not generated until these valrefs are known. So the list is sorted.
- A new valref is created for every ALICE macro, and these valrefs can be created in order.

Termination can be made plausible by the following considerations:

- Steps 1, 2 and 3 of *obtaining-a-valref-for-a-defref* (5.1.2.2.1.3) terminate immediately. Step 4 asks for the processing of an unmarked definition.
- Step 1 of *processing-a-definition* (5.1.2.2.1.2) marks the definition. All its steps terminate immediately, except those calling for *obtaining-a-valref-for-a-defref*.
- For each application of 5.1.2.2.1.3 followed by 5.1.2.2.1.2, an unmarked definition gets marked. Since the number of definitions is finite, this process terminates.
- Each definition marked in 5.1.2.2.1.2.1 will be removed in 5.1.2.2.1.2.6. So the driver will also terminate.

5.1.2.2.5. Alternative algorithms

5.1.2.2.5.1. Sorting

Any topological-sort algorithm can be used. An algorithm that suggests itself scans the list of definitions and tests each definition for dependency on definitions which have not been processed yet. If it does not depend on such definitions, it is processed and an ALICE-value is generated. This process is continued until no further progress is made. If there are unprocessed definitions left, they are in error or depend on erroneous definitions. A separate algorithm is needed to disentangle this knot and give reasonable error messages.

The algorithm may be useful if memory is very much limited since it allows much data to be kept on backing store. While scanning the definition list it can produce a new definition list plus some ALICE-values and subsequently scan this new list. Information about whether or not a definition has been processed can be obtained from the translation table.

5.1.2.2.5.2. Counting

We could keep track of the number of values while producing the definitions, rather than counting them in *read-values-into-direct-access*. This has the advantage that the number will be available at the right moment, and *create-status-information* and *create-values* can be executed in their proper order, whereas now they have to be

merged. The disadvantage is that the counting is distributed over the entire reading process; this is unreasonable since the validity of the counting depends on the sorting algorithm.

5.1.2.2.6. Conclusion

This concludes the implementation-independent design of *sort-and-count-and-output-values*, and therewith that of *create-values*.

5.1.3. Further design, stages 1 & 2

Create-data, *create-communication-area* and *create-rules* (5.1) have been designed along the lines demonstrated above, and have been used as a basis for stage 3 of the design (5.2.3). Since they consist of nothing but more details, they are not presented here.

5.2. Obtaining and organizing the information

Now that we know exactly what information we need for every construct in the language in order to translate it, we shall turn to devising ways of obtaining and organizing this information.

Detailed information is necessary for *create-values*, *create-data* and *create-rules*, and this information is interrelated through tags, defrefs, an information aggregate called declaration-info, alternative graphs, statement graphs, symbolic run-time stacks, (the last three of which occur in the design of *create-rules* which is not given here), etc. We shall describe here only the data-manipulation required for generating ALICE **values**.

5.2.1. The tag-list

In the description in 5.1.2 tags and defrefs are continually looked up, but it would of course be ridiculous to do so in actual practice. A tag occurring in the program is looked up in a tag-list once and is then replaced by a pointer to the entry in the tag-list. Thereafter the pointer gives immediate access to the information needed and no further searching is necessary.

We shall now see how this is done in more detail. When we meet a tag in the program text, it is one of the following:

a selector,
a formal or local,
a global (constant, variable, rule, etc.) or
an undefined tag.

In each of these cases the sequence of characters of the tag must be saved for posterity: the formals or locals for the dump-pragmat, and undefined tags for error-messages. So the tag is looked up in one big list of strings, and when in the sequel we speak of a tag we mean the pointer to this string. We first check (from immediate context) if it is used as a selector (which is saved until we see the list tag). Next we check if it is a formal or local; if so, we treat it as such. If not, it is a global tag.

We may not have seen its declaration yet, or its declaration may be missing, or there may be multiple declarations for it. So we are tempted just to replace the tag by the pointer we have in our hands, since this is all we know. But that would defeat

our purposes: the next time somebody gets hold of this tag (i.e., this pointer) he wants information about it, e.g., where it occurred or how and where it was declared. So the replacement pointer must be to a global-info-block containing the following information:

a pointer to a string (just obtained),
a pointer to declaration-info (initially empty),
a pointer to cross-reference info (initially to the present occurrence),
marking bits.

The declaration-infos can be different for different types of declarations, since information of a different nature must be stored for each. The cross-reference information could be a chained list of line numbers, which need not be kept in direct access. The tag list and the global-info-list will have to be present all the time.

We can now see a global tag as a pointer to a global-info-block containing information about, e.g., its string. This information is unreliable until the entire program has been read, and may be so even thereafter if the program is wrong.

Since some information which is independent of the declarations must already be collected at an early stage (see, e.g., 5.1.2.1.2.2), room for marking bits is supplied in the global-info-block.

The actual compiler data structures and their interrelations are described by F. van Dijk [VAN DIJK 82].

Implementation Note:

The tag-list algorithm used in the compiler is the one described in [GRUNE 77]. Pointers to global-info-blocks are kept on a stack, stored in the order of the strings in the global-info-blocks. This data structure allows binary search; the insertion problem is solved by keeping the stack diluted with *nil*-pointers, which can be sacrificed upon insertion of a new tag. Redilution takes place when the percentage of *nil*s sinks below a given minimum value (about 4 percent).

5.2.2. *Create-values*

As we know, *create-values* consists of two phases, one collecting 'definitions' and one sorting these definitions into ALICE-values, meanwhile producing a translation table. The definitions are in essence produced sequentially so that hopefully they can be written to a file, which would lower storage requirements (4.2.1).

Since these definitions form the interface between the two phases, we are tempted to tackle these definitions first and choose a (language-independent) representation for them, so that *collect-values* will know what to produce and *sort-and-count-and-output-values* will know what to expect. The grammar of these definitions is given in 5.1.2.1.10; it is full of 'defrefs' the grammar of which is also given there. We should therefore design representations for these defrefs, but in doing so we are confronted with a bewildering variety of forms and the question arises whether *collect-values* really has to produce such complicated things. The possible forms are:

sumsize!TAG,
virtsize!TAG,
virtleftover!TAG,
sizeleftover!TAG,
#[N]TAG,
TAG,
!<TAG, *<<TAG*, *<>TAG*, *>>TAG*, *>!TAG* and
#[N].

It would be nice to let the first four coalesce into a single *#[N]TAG* or, better still, the first five into a *#[N]*. However, this must not make the error messages worse.

The definition list (or definition file) serves to pass information to *sort-and-count-and-output-values* and it should be in such a form as to do so effectively. This means that the format should be such that the usual operations on the list are simple and cheap. Now the algorithm in 5.1.2.2.1.3 requires finding the definition of a given defref *DR*, and if no care is taken, this could be an expensive operation.

If the defref involves a *TAG*, this is a tag from the program (or *#FL*, *#AST* or *#RST*, which, if need be, could be simulated), and we can expect that a definition can be found through the tag-list and the global-info.

If, however, the defref is *#[N]*, it is just an integer and in principle we have to search the definition list to find its definition. But if it is 'just an integer' we could try to let it be a reference to the position of its definition, e.g., the serial number of that definition. This is, of course, only possible if the serial number of a definition of a *#[N]*-defref is always known by the time the defref is used in another definition. At first sight this seems to be true; we shall have to verify this in the design of *collect-values*.

We can summarize our wishes for the definition list as follows. Definitions come in the following forms (5.1.2.1.10):

operator:	operands:
=	*defref defref*
{ =+, =−, =×, =/ }	*defref defref defref*
=*intdenot*	*defref string*
=*chardenot*	*defref character*
=*strlength*	*defref integer*
=*manfcon*	*defref symbol*
=*extsize*	*defref string*

and defrefs come in three forms:

TAG
!<TAG, *<<TAG*, *<>TAG*, *>>TAG*, *>!TAG*
#[N]

If a defref of the form *#[N]* occurs as a first operand (i.e., is being defined), that definition must be the *N*-th definition.

These design requirements do not follow logically from anything said so far. They are tentative additional requirements made for the sake of efficiency, of which we hope that they will not lead us into trouble elsewhere. If they do we shall have to back up and review the situation.

5.2.2.1. *Collect-values*

Collect-values addresses itself to

constant-descriptions,
table-heads,
stack-heads,
filling-list-packs,
pointer-initializations,
constant-sources, and
expressions.

5.2.2.1.1. Constant-descriptions

A **constant-description** equates an ALEPH **tag** to an **expression**. The **expression** is processed, which yields a defref. If this defref is not of the form *#[N]*, a definition for the next secret defref *N* is generated, to be equal to the given (named) defref. The **tag** is looked up in the tag-list. If the declaration-info is empty, it is now set to the triplet

(CONSTANT, line number, N);

otherwise there is a double definition.

This declaration-info provides easy access to the tag's definition in the sorting phase.

5.2.2.1.2. List-heads

A list-head defines a list identified by a tag. This tag is looked up in the tag-list. It may already have a non-empty declaration-info, in which case an error message is in order. In essence no definitions are generated then, but we must keep in mind that some **pointer-initialization** may depend on this faulty declaration.

If the tag is still 'free', a declaration-info of some form must be appended. It should allow easy access to the definitions of various **limit**s, preferably in the form of the serial numbers of their definitions. ALICE requires for its **list-info** of a list *L* the virtual-min-limit *!<L*, the virtual-max-limit *>!L*, the min-limit *<<L*, the max-limit *>>L*, and the calibre *<>L*, despite the confusing terminology in ALICE Manual 3.2.2.2 (see also 5.1.2.1.4). A declaration-info of the following form seems reasonable:

(TABLE/STACK, line number, !<LST, >!LST, <<LST, >>LST, <>LST).

The final value of the max-limit field will be set during the processing of the **filling-list-pack** since it cannot be correctly set earlier. There is a variable *prev lst* (5.1.2.1.4.2) which refers to the name of the previous list of the same type as the present one. As we see from the last paragraph of 5.1.2.1.4.2, we shall also have to keep track of *virt size*.

We shall now look into the details.

5.2.2.1.3. Table-heads

When we read 5.1.2.1.4.1.1 we are immediately confronted with two problems: *last fixed* and the definition $<>FL = CAL$. CAL is a genuine integer and integers are normally handled in string form only. We can do one of two things now. Either we convert CAL into a string and produce

$=intdenot \qquad <>FL \;\; CAL-string$

or we introduce a new type of definition and produce:

$=int \qquad <>FL \;\; CAL \;.$

The latter seems simpler. It does not make any difference for the ALICE code, since both would result in:

$INT\ valref,CAL\ .$

The *last fixed* causes more problems. It introduces an inconvenient tag $\#FL$ which should presumably be entered in the tag-list with the definition of some table prior to all other tables. But if we look more closely we see that only $>!\#FL$ is used: it gives the value of the virtual right limit of the zero-th table and at the end of the program it is set to the minimum virtual address minus 1 (ALICE symbol *MNA*). So a single tag suffices.

But there is no reason to postpone the initialization of $>!\#FL$ to the end of the program. We can start by making a new defref $\#[N]$ and generate a definition:

$=manfcon \qquad \#[N] \;\; MNA \;.$

This suggests that *last fixed* can be represented by an integer variable *N last fixed* such that:

$$\#[N\ last\ fixed] \;=\; >![last\ fixed]\ .$$

The same applies to $>![prev\ lst]$ which turns into $\#[N\ prev\ lst]$. This brings us to the following actions.

For each **table-head** with tag FL and calibre CAL we obtain four new secret defrefs $N1$ to $N4$ and generate the following definitions:

$=+ \qquad \#[N1] \;\; \#[N\ last\ fixed] \;\; \#[N2]$
$=int \qquad \#[N2] \;\; CAL$
$=+ \qquad \#[N3] \;\; \#[N\ last\ fixed] \;\; \#[N4]$
$=- \qquad \#[N4] \;\; >>FL \;\; \#[N\ last\ fixed]$

If the tag is still free, a declaration-info of the form

(TABLE, line number, N1, N3, N1, N last fixed, N2)

is appended to it. *N prev lst* is set to *N last fixed*, *N last fixed* to *N3* and *N virt size* to *N4*.

In the case of an external-table with string *STR* the last definition is replaced by

$=extsize \qquad \#[N4] \;\; STR$

and the first entry in the declaration-info is EXTERNAL rather than TABLE.

Remarks:

- The definitions for *N3* and *N4* together calculate the max-limit and the virtual-size. The form of the definition of *N3* is chosen to match those in 5.2.2.1.5 and 5.2.2.1.6.
- The variable *last fixed* which refers to a stack or a table has been replaced by *N last fixed* which refers to an integer.
- The $>>FL$ field has been set provisionally to *>![last fixed]*, the value that should result from a missing or bad **filling-list-pack**.

5.2.2.1.4. Stack-heads without size-estimate

These are treated like **table-head**s except for the declaration-info which will be:

(STACK, line number, N1, N3, N1, N last fixed, N2).

5.2.2.1.5. Stack-heads with absolute-sizes

Again the question arises what to do about $\#AST$. As before only $>!\#AST$ is ever used but its definition

$$>!\#AST = >![last\ fixed]$$

cannot be generated until the very end of the program. So here we have the problem in full bloom and there seems to be no way out but to introduce a secret tag $\#AST$, generate a definition in the beginning

```
=          #[N]  #AST ,
```

and use *N* as starting value of *N last ast*. At the end of the program we then act as if we had seen an ALEPH **constant-description** for $\#AST$, which results in the declaration-info of the form

(CONSTANT, line number, N last fixed)

to be appended to it.

This is not too messy a solution, since $\#AST$ is not really a tag but only a pointer to a global-info-block (5.2.1) which may have NIL for pointer-to-string. So in scanning the tag-list we will never meet it.

The processing of an absolute-size **stack-head** is then straightforward. If the *SIZ* **expression** is not of the form *#[N]*, we generate an intermediate definition to make it so. We then grab three secret defrefs *N1* to *N3* and generate

```
=+         #[N1]  #[N last ast]  #[N2]
=int       #[N2]  CAL
=+         #[N3]  #[N last ast]  SIZ
```

If the tag is still free, a declaration-info of the form

(STACK, line number, N1, N3, N1, N last ast, N2)

is appended to it. *N prev lst* is set to *N last ast*, *N last ast* to *N3* and *N virt size* to *SIZ*.

5.2.2.1.6. Stack-heads with relative-sizes

When we read 5.1.2.1.4.1.3 we meet *#RST*, which could be handled in the same fashion as *#AST* as far as its *>!#RST* aspect is concerned. The text, however, mentions various other defrefs to be attached to an *RST* tag and consequently to *#RST*. These other defrefs are

sumsize!RST,
virtsize!RST,
virtleftover!RST and
sizeleftover!RST.

Of these, *virtsize!RST* is used for local calculations only; the others are used locally and in one other place: the description of the next relative-size stack. So they need not be stored with the declaration of the present relative-size stack and can remain global, to be used and then reset by the next relative-size stack description.

We shall need four globals, *N last rst*, *N last sumsize*, *N last sizeleftover* and *N last virtleftover*, such that:

#[N last rst] = >![last rst],
#[N last sumsize] = sumsize![last rst],
#[N last sizeleftover] = sizeleftover![last rst], and
#[N last virtleftover] = virtleftover![last rst].

Their initializations can be achieved by a combination of existing tricks:

- *N last rst* starts as the number of a definition equating it to the pointer to a global-info-block of a secret tag *#RST*, which at the end of the program will be set according to a **constant-description** equating that tag to *#[N last ast]*; in other words, it is 'indirectly initialized' to *N last ast*.
- *N last sumsize* starts as the number of the definition of 0 (which is 0 (5.1.2.1.9)),
- *N last sizeleftover* is 'indirectly initialized' to *N last sumsize*, as with *N last rst* above,
- *N last virtleftover* is likewise 'indirectly initialized' to a secret defref *N1* for which the definition

  `=-` *#[N1] manifest constant(MXA) #[N last ast]*

  is generated.

Processing a relative-size **stack-head** is then done as follows. If the **expression** in the **relative-size**, *SIZ*, is not of the form *#[N]*, it is made to be so. We then grab eight secret defrefs *N1* to *N8* and generate

```
=+      #[N1] #[N last rst] #[N2]
=int    #[N2] CAL
=+      #[N3] #[N last rst] #[N4]

=×      #[N4] #[N5] SIZ
=/      #[N5] #[N last virtleftover] #[N last sizeleftover]
=-      #[N6] #[N last virtleftover] #[N4]
=+      #[N7] #[N last sumsize] SIZ
=-      #[N8] #[N last sizeleftover] SIZ
```

If the tag is still free, a declaration-info of the form

(STACK, line number, N1, N3, N1, N last rst, N2)

is appended to it. Set

N prev lst to *N last rst*,
N last rst to *N3*,
N virt size to *N4*,
N last virtleftover to *N6*,
N last sumsize to *N7*, and
N last sizeleftover to *N8*.

5.2.2.1.7. Filling-list-packs

When reading 5.1.2.1.4.2 we see that the counter *n* is no longer necessary since its actions are covered by the general creation of secret defrefs. *N last pointer* is initialized to *N prev lst* and *offset* is initialized to 0.

For **single-** or **compound-block**s *offset* is increased by *CAL*.

For each **string-denotation** of length *K* we 'update' *N last pointer* (see below). We then process the increase caused by the **string-denotation**. We grab two defrefs *N1* and *N2*, generate

```
=strlength      #[N1]  K
=+              #[N2]  #[N last pointer]  #[N1]
```

and set *N last pointer* to *N2*.

N last pointer is "updated" as follows: if *offset* equals 0, *N last pointer* is already updated; if not, we grab two defrefs *N1* and *N2*, generate

```
=int            #[N1]  offset
=+              #[N2]  #[N last pointer]  #[N1]
```

and set *N last pointer* to *N2* and *offset* to 0. (The new *=int* operator comes in handy here.)

For each **pointer-initialization** with **tag** *PNT* we update *N last pointer* and append the declaration-info

(CONSTANT, line number, N last pointer)

to the tag.

Finally, at the end of the **filling-list-pack** we update *N last pointer* as described above and set the max-limit field of the list declaration-info to *N last pointer*.

For an absolute-size stack with tag *AST* a definition of the form

```
=-              #[N1]  #[N last ast]  #[N last pointer]
```

is generated to indicate the number of 'fallow' addresses in the **list-area** (ALICE Manual 3.2.2.1).

The defref indicating the number of virtual addresses is *#[N virt size]*; the corresponding valref is needed for the ALICE **list-area**.

Note that the updating of *N last pointer* implements the optimization for *offset* 0 as described in 5.1.2.1.9.

5.2.2.1.8. Expressions

Expressions consist of **terms**, **bases** and **plain-values**. For **terms** and **bases** we generate straightforward definitions as described in 5.1.2.1.6, 5.1.2.1.9, and 5.1.2.1.10. None of these will ever use a secret defref of which the definition has yet not been produced. **Plain-values** come in four kinds:

integral-denotations,
character-denotations,
constant-tags and
table-**limits**.

Constant-tags and table-**limits** are themselves defrefs (they might be undefined or misdefined). **Integral-** and **character-denotations** produce definitions of the form

=intdenot	*#[N] string*
=chardenot	*#[N] character.*

5.2.2.1.9. Constant-sources

Constant-sources are **plain-values**; see above.

5.2.2.1.10. The grammar of the definition list

Definitions come in the following forms:

operator:	operands:
=	*#[N] defref*
{ =+, =−, =×, =/ }	*#[N] defref defref*
=int	*#[N] integer*
=intdenot	*#[N] string*
=chardenot	*#[N] character*
=strlength	*#[N] integer*
=manfcon	*#[N] symbol*
=extsize	*#[N] string*

and defrefs come in three forms:

TAG
<<TAG, <>TAG, >>TAG
#[N]

This grammar completely satisfies the requirements formulated in the last few paragraphs of section 5.2.2 (except for the new operator *=int*, the processing of which is trivial). It even exhibits two more properties that might be utilized. We see that the defref to be defined is always of the form *#[N]*, and we know that this *N* is the serial number of the definition. This means that the *#[N]*s are superfluous. If we leave them out, the definitions turn into expressions, which fact we can emphasize by also omitting the =-sign from the operator. The *N*s in defrefs, in declaration-infos and in the intermediate (5.1.2.1.7) text must then be regarded as expression numbers.

It seems, however, inadvisable to change our terminology at this point. We shall therefore continue to call definitions definitions and leave the =-sign in.

Moreover, the forms *!<TAG* and *>!TAG* no longer occur as defrefs. It is satisfying to see that the notion 'defref' exactly reduces to the notion '**plain-value**', where the *#[N]* originates from naming **integral-denotation**s and **character-denotation**s.

5.2.2.2. *Sort-values*

Little needs to be added in this stage to the algorithm described in 5.1.2.2.

The elements of the translation table are of the form *(defref, valref)*, where the *defref* is defined in a definition. We know now, however, that all defrefs defined in definitions are of the form *#[N]*. This suggests that the translation table can be kept as a consecutive list of valrefs, their positions in the list providing the defrefs. (The table is used exclusively for translating defrefs into valrefs, not vice versa). Since valrefs start from 1, 0 can be used to indicate that the corresponding definition has not yet been processed (step 5.1.2.2.1.3.1).

In 5.1.2.2.1.2.1 a definition is marked UNDER CONSIDERATION; thereafter some actions occur which result in an entry in the translation table and the removal of the present definition. Since the position of a definition has become relevant, this removal cannot be taken seriously. A new mark REMOVED might be introduced but it appears that the UNDER CONSIDERATION mark can figure as such:

- If a definition is marked UNDER CONSIDERATION (step 5.1.2.2.1.2.1) its entry in the translation table will certainly be filled (5.1.2.2.1.2.5) and it will certainly be removed (5.1.2.2.1.2.6).
- If the entry for a definition is filled (5.1.2.2.1.3.1) its UNDER CONSIDERATION mark will not be examined (5.1.2.2.1.3.3).

In other words, as soon as the entry is filled, the UNDER CONSIDERATION mark ceases to have a meaning.

Incorrect definitions are replaced by a valref of zero, which originates from the definition (5.1.2.1.9)

#0 = 0 .

We now arrive at the following algorithms.

5.2.2.2.1. The reader

It requires two counters, *number of defrefs* and *number of valrefs*, both starting at zero. For each definition in the definition list (actually on the definition file) a definition of the form

(FALSE, operator, operand1, operand2)

is created and *number of defrefs* is increased by one. If the operator is not =, *number of valrefs* is also increased by one.

When all reading is done, a translation table is created with *number of defrefs* entries, all zero.

5.2.2.2.2. The driver

The algorithm has a global variable *last tag* (for error messages only), initially set to *nil*. As long as the definition list still contains a definition of which the first field is FALSE, process that definition.

Otherwise, the process is finished and the definition list can be discarded.

5.2.2.2.3. Processing a definition *D* with serial number *N*

1) set the first field of *D* to TRUE.
2) if the operator is '=', obtain a valref *v1* for *operand1*.
   3.1) if the operator is '=*int*', make a new valref *v1* and generate

        INT v1,operand1

   3.2) similar actions for '=*intdenot*'.
   3.3) similar actions for '=*chardenot*'.
   3.4) similar actions for '=*strlength*'.
   3.5) similar actions for '=*manfcon*'.
   3.6) similar actions for '=*extsize*'.
   4.1) if the operator is '=+', obtain valrefs *v2* and *v3* for *operand2* and *operand3* respectively, make a new valref *v1* and generate

        ADD v1,v2,v3

   4.2) similar actions for '=−'.
   4.3) similar actions for '=×'.
   4.4) similar actions for '=/'.
5) set the *N*-th entry in the translation table to *v1*.
6) yield the valref *v1*.

5.2.2.2.4. Obtaining a valref *V* for a defref *DR*

1) if *DR* is of the form *TAG* set *last tag* to *TAG* and check if the *TAG* has a declaration-info the first field of which is CONSTANT. If so, set *M* to the third field; otherwise, give an error message with *last tag* and line number, and set *M* to the defref of zero,
2) if *DR* is of the form *<<TAG*, *<>TAG* or *>>TAG*, set *last tag* to *TAG* and check if the *TAG* has a declaration-info the first field of which is TABLE. If so set *M* to the indicated field; otherwise give an error message with *last tag* and line number and set *M* to the defref of zero,
3) if the *M*-th entry in the translation table is non-zero, yield the valref found there;
4) otherwise, consider the *M*-th definition;
   4.1) if its first field is TRUE, give an error message with *last tag* and present line number, and yield the valref of zero;
   4.2) otherwise, the definition is proper; process it and yield the valref thus obtained.

5.2.2.2.5. Conclusion

This concludes the information-collecting phase of the design of *sort-values* and therewith that of *create-values*.

5.2.3. Further design, stage 3

As in 5.1.3, the stage 3 design results for the rest of the compiler are not presented in this thesis.

6. MODIFICATIONS TO ALICE

6.1. Inconsistencies in the ALICE definition

ALICE is defined three times in the ALICE Manual [BÖHM 77]; once in paragraph 2.5, where a regular grammar is given which produces the ALICE-macros in any order, regardless of their interrelationship; once in paragraphs 3.1 to 3.4, which contain a context-free grammar interspersed with semantics and explanations; and once in paragraph 3.5, where all bits of grammar from paragraphs 3.1 to 3.4 are collected into one grammar. All three definitions differ in small points; these differences do not impair the understandability. For implementation, however, it is necessary that there be one grammar.

The regular grammar was disregarded since the implementation was based on a context-free grammar (for error-checking purposes). Fortunately the context-free grammars complemented each other. The numerous inconsistencies in names (e.g., **ext-table-decl** is sometimes called **external-table-decl**) were solved in favour of the shorter name. All declarations missing from the distributed grammar (e.g., those for **values**, **data**, **list-type**, **sp**, etc.) could easily be supplemented. The few remaining errors were solved as follows.

- **Output-gate-creation** is obligatory in ALICE Manual 3.3.5 and optional in ALICE Manual 3.5. It was made obligatory for two reasons:
  - Its mirror image **input-gate-creation** is obligatory in both grammars.
  - It is the philosophy of ALICE to be as explicit as possible, so it is better to indicate an empty gate by creating one of size 0 than by not creating it. A macro processor for ALICE will benefit from this.
- The exit-value in **exit** in ALICE Manual 3.5 is specified as a **valref** only: the **repr** is missing. This is wrong: since the value must be accessible at run-time, it must be addressable through a **repr** and there must be a **constant-source** for it.

These changes resulted in a grammar which was declared *the* context-free grammar intended in the ALICE Manual. The further sections in this chapter treat shortcomings of and modifications to this grammar.

6.2. Shortcomings of ALICE

In the course of the design of the compiler a number of difficulties with ALICE were observed. Most of these were very easy to correct, but four problems required further investigation.

Minor points included:

- The standard externals *set elem* and *string length* had the same internal representation *STL*; *set elem* was renamed *SEL*.
- Some symbols were missing, e.g., the one to be used in the translation of a **transport**.
- Everything connected with external constants was missing.
- The grammar (inadvertently) did not allow an **ext-table-decl** when there is no **list-area**. In a first attempt to correct this, all components of **lists** were made optional (the **[]** indicate optionality):

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

However, since **data** is defined:

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

there are now two ways to describe the absence of lists and the grammar is ambiguous. A correct solution is obtained by treating the components of **lists** on the same footing as those of **data**:

data:
[constant sources],
[variable decls],
[list areas],
[ext table decls],
[list administrations],
[file administrations].

which also rids us of a superfluous rule **files**.

- It was found unrealistic to keep the user-pragmats (and comments) out of the formal grammar of ALICE.
- A number of symbols were missing from **extag** (i.e., from the list of standard externals), e.g., for *delete*, *unqueue to*, etc.
- Only those constants that do not get their values in the ALEPH program or postlude need to have a symbol as a **manifest-constant**. Thus, no symbol is required for TRUE, FALSE, etc.
- There is a slight irregularity in the definition of the **unstack-and-return** macro. It is the only macro that is directly generated in more than one place, and used with more than one meaning (in **unstack-and-return-true** and **unstack-and-return-false**). The distinction is made by a parameter (**true-symbol** versus **false-symbol**): this is the only place where the grammar prescribes a fixed parameter. The anomaly is solved by splitting the **unstack-and-return** macro into two.
- The translation of a 'dummy' affix (ALEPH Manual 3.4) requires the **store-w_reg-sequence** in **restore-from-output-gate** to be optional.
- Some machines allow a more efficient calling sequence for non-recursive calls than for recursive ones. In such cases the hardware places the return information in a fixed place somewhere near the **rule-head**. The 'success tail/fail tail' must have access to it, so the corresponding macros need the repr of the rule.

The four more serious problems are discussed in the following paragraphs.

6.3. ALICE is not of type LL(1)

There is no direct reason why the grammar of ALICE should be an LL(1) grammar. The stream of ALICE macros is intended to be processed macro by macro, in a finite state fashion; and the regular grammar of the macro stream is clearly of type LL(1), since each macro is identified by a unique initial symbol.

There are, however, good indirect reasons for the grammar to be of the type LL(1).

- It allows the ALICE processor to parse easily the macro stream according to the context-free grammar. In this way the circumstances of each occurrence of each macro are known, which can be useful for code optimization.
- It is advantageous during development to be able to check the ALICE stream against its context-free grammar.

Fortunately the ALICE grammar is almost of type LL(1). The only problem is caused by production rules starting with **load-addr-in-a_reg**.

- It is not possible (on an LL(1)-basis) to determine the presence (or length) of the **load-list-element-in-v_reg-sequence** in **store-w_reg-in-list-element**.
- It is not possible to distinguish between **copy-val-to-input-gate** and **copy-addr-to-input-gate** in **copy-to-input-gate**.
- It is not possible to discern the end of **copies-to-input-gate** in **extension**.

We shall now treat the first two problems; since the grammar and semantics of the **extension** in ALICE Manual 3.3.8.2.3 are clearly incomplete, the treatment of the third problem is better combined with the design of a correct **extension** sequence (6.5).

The grammar of **load-indexed-element-in-v_reg** and its complement **store-w_reg-in-indexed-element** (incorrectly named **store-w_reg-in-list-element** in the ALICE Manual) is not as clean as would be desirable. It causes implementation problems for the implementer who wants to use registers for the gate and a subroutine for index checking and indexing. The implementer then has the choice either

- to identify v_reg with the machine register which holds the top of the gate, and have several different subroutines for indexing via the various gate registers, or
- to identify v_reg with a fixed machine register, known to the indexing routine, and fill the gate register afterwards.

Neither of the alternatives is really attractive. The problem clarifies when we introduce, just for the sake of argument, an index register i_reg. An indexed input parameter with n (nested) indices could then produce:

load simple in i_reg,

followed by $(n-1)$ times

load addr in a_reg,
load i with list elem from i_reg,

followed by

load addr in a_reg,
load v with list elem from i_reg.

A similar output parameter would need:

load simple in i_reg,

followed by $(n-1)$ times

load addr in a_reg,
load i with list elem from i_reg,

followed by

load addr in a_reg,
store w_reg in list elem under i_reg.

This approach provides the user with exact information about which register to use. A practical disadvantage is that it requires part of the grammar to be duplicated with i_reg instead of v_reg. To avoid this we introduce a symbol **index-symbol** with the meaning: from now on all references to v_reg actually reference i_reg. The symbol **end-index-symbol** switches this interpretation off. We then get:

load indexed element in v_reg:
load index sequence,
load list element in v_reg.

store w_reg in indexed element:
load index sequence,
store w_reg in list element.

load index sequence:
index symbol, el,
load simple in v_reg,
[load list element in v_reg sequence],
end index symbol, el.

store w_reg in list element:
load addr in a_reg,
store w list element symbol, sp, integer, el.

A similar reasoning applies to **copy-addr-to-input-gate**. Normally a_reg is used to access objects, but here it only serves as an intermediate register for an address on its way to the gate, a function for which v_reg might be more appropriate. It seems fair to indicate this odd usage of a_reg to the implementer:

copy addr to input gate:
copy address symbol, el,
load addr to a_reg,
copy a_reg to input gate.

This also solves the first two LL(1) problems.

6.4. The calling mechanism

The parameter passing in ALICE is described in terms of an (abstract) gate, onto which the input parameters are loaded by the caller, from which they are fetched by the called rule, onto which the called rule writes its output parameters, and from which the caller extracts the results. The details are such that the system supports two implementation techniques, one in which the role of the gate is played by registers (to be called 'scheme *A*') and one in which the gate is mapped directly on the correct

positions in the stack frame of the called rule ('scheme *B*').

Scheme *A* works perfectly, but scheme *B* causes problems. In order to understand why this is so we have to look at the information necessary for implementation. For each scheme we shall consider four items: the call of rule *S* in rule *R*, the rule head of *S* ('rule entry'), the rule tail of *S* ('rule exit'), and the restore by the caller in *R*.

For scheme *A* we have:

call of *S*:
    some values → gate registers,
    link to rule *S*.

rule head of *S*:
    allocate formals and locals of *S*,
    gate registers → some formals.

rule tail of *S*:
    some formals → gate registers,
    deallocate formals and locals of *S*,
    unlink to caller.

restore in *R*:
    gate registers → some locations.

(What return information is provided in the 'linking' to the rule and where it is stored is left unspecified here, under the proviso that it can be used in the 'unlinking' to the caller.)

For scheme *B* we get:

call of *S*:
    allocate formals and locals of *S*,
    some values → some formals of *S*,
    link to *S*.

rule head of *S*:
    empty.

rule tail of *S*:
    unlink to caller.

restore in *R*:
    some formals of *S* → some locations,
    deallocate formals and locals of *S*.

It appears that *R* has to know the number of locals of *S*, as they are indeed provided in the **target-stack-frame**-macro (ALICE Manual 3.3.6). The ALEPH compiler, however, cannot reasonably provide this information:

- The calculation of the number of locals is a tricky affair, since implicit locals may be needed (e.g., to implement the 'spoil and fail' effect described in ALEPH Manual 3.7). The presence of implicit locals can only be detected when the rule is fully analyzed, which may be after the call. The problem can be solved, but only at the expense of another pass over the text.
- If the call is to a separately compiled rule, the number of locals is unknown. Now separate compilation is not a feature of ALEPH as described in the ALEPH Manual, but it would be nice to add it in a simple form, and if calls to locally and separately compiled rules differ too greatly, complications arise.

It has been suggested that the problems with scheme B can be solved by having the caller allocate the formals only (and fill them as need be). The locals will then be allocated by the called rule. A consequence of this is that each calling sequence involves 2 allocations and 2 deallocations, which seems exaggerated.

This technique, however, allows a simple optimization. If the maximum number of formals ever to be allocated in any 'call of X' in R is known in advance, the necessary space can be allocated in the rule head of R once and for all. These location are called the 'actuals' of R.

The calling sequence is then (scheme C):

call of S:
    some values $\rightarrow$ actuals of R,
    link to S.

rule head of S:
    allocate locals and actuals of S
        (actuals of R $\Rightarrow$ formals of S).

rule tail of S:
    deallocate locals and actuals of S
        (formals of S $\Rightarrow$ actuals of R),
    unlink to caller.

restore in R:
    actuals of R $\rightarrow$ some locations.

(The symbol $\Rightarrow$ is used to denote 'reinterpretation', as opposed to $\rightarrow$ which means 'copying'.)

We are now in a position to reassess the information needed in the four steps.

At the call we need the number of input parameters.

At the rule head we need (number of formals + number of locals) and (number of locals + number of actuals).

At the rule tail we need the same plus the number of output parameters.

At the restore we need nothing.

This means that the **target-stack-frame** disappears from the **call** sequence. If we now introduce an **input-gate-creation**-macro in **ext-call**, the parameter treatment in **call** and **ext-call** is sufficiently similar that **ext-call** can be used for a call to a separately compiled rule.

There is one place where a call occurs outside a **rule-body**, viz., in the **root**, as the initial call by the main **program**. The **root** must set up an environment equal to that

of a normal rule, so it must be given information about the number of parameters (always 0), the number of locals (also always 0) and the number of actuals (equal to the number of parameters in the **affix-form** in the **root**). The **root**-macro has been extended to this effect.

It should be noted that scheme *C* produces less code than scheme *B*: there is one allocation/deallocation for each rule rather than for each call.

Some thought has been given to machines on which indirect addressing is cumbersome and undesirable. On such machines one would like to place the formals and locals of each rule in fixed locations. This causes no problem if the rule is non-recursive (scheme *D* 1):

call of *S*:
    some values → input formals of *S*,
    link to *S*.

rule head of *S*:
    empty.

rule tail of *S*:
    unlink.

restore in *R*:
    output formals of *S* → some locations.

If, however, *S* is recursive, the formals may be occupied already, and the use of a gate is unavoidable (scheme *D* 2):

call of *S*:
    some values → gate,
    link to *S*.

rule head of *S*:
    if formals of *S* in use: formals and locals → stack,
    gate → input formals of *S*.

rule tail of *S*:
    output formals of *S* → gate,
    if formals and locals stacked: stack → formals and locals,
    unlink.

restore in *R*:
    gate → some locations.

It should be noted that almost any other conceivable parameter passing mechanism can be implemented by having the assembler store the necessary information before the rule entry, after the rule exit or at the program end, and picking it up dynamically.

It is interesting to see that in a certain sense the 3 above schemes *A*, *B* and *C* are the only ones. If we assume that any calling sequence must consist of the following 5

indivisible actions:

a: values → gate
b: link
c: allocate formals
d: allocate locals
e: gate → formals,

then there are 120 possible permutations. Now e cannot occur before a, nor before c (gate or formals not yet available); and b cannot precede a (values no longer available). This reduces the number to 25. For reasons of efficiency we are now interested in subsequences that can be contracted. There are 3 such subsequences:

$ae \Rightarrow$ values → formals,
$cd \Rightarrow$ allocate formals and locals,
$dc \Rightarrow$ allocate locals and formals.

We realize that cd and dc are essentially the same, which lowers the number of different sequences to 20. This gives the following table:

number of sequences:	20	
number of sequences with ae:	5	
number of sequences with cd:	3	
number of sequences with both:	1	(scheme $B = cdaeb$)

Up to now we have neglected the problem, explained above, that the number of the locals required is not available before the actual linking, so that d cannot precede b. Introduction of this restriction changes the picture drastically:

number of sequences:	8	
number of sequences with ae:	1	(scheme $C = caebd$)
number of sequences with cd:	1	(scheme $A = abcde$)
number of sequences with both:	0	

The remaining 6 sequences (*abced*, *acbde*, *acbed*, *acebd*, *cabde* and *cabed*) are mostly stupid variations of scheme A or C; anyway, they do not contain any interesting contractible sequences.

6.5. The extension sequence

The **extension** sequence as stated in ALICE Manual 3.3.8.2.3 causes immediate problems both for the compiler writer and for the implementer.

- What values should the compiler generate for the formals in **copies-to-input-gate**?
- What code should be generated for an ALEPH **extension** in which one **source** is **transport**ed to more than one **selector**?
- How can the **extension** sequence be implemented on a machine without registers, i.e, under scheme C? At best the implementer is forced to allocate the gate somewhere in memory, as if it consisted of registers.

To gain a better insight in the problem we shall write down the steps needed under scheme A and C. For scheme A we have:

values → gate,
addr of stack administration → a_reg,
extend stack and update stack administration (using a_reg),
gate → stack block
(including multiple **transports**, using a_reg).

This is the basic sequence as supported by the ALICE Manual. It corresponds closely to the semantics (ALEPH Manual 3.4.3): first the values are calculated, then the stack is extended, then the block obtained is filled.

If we have, however, a machine without registers, and want to avoid a simulated gate in memory, the calculated values must be stored directly in the stack block, which must be available by then. Note that the max-limit of the stack should not yet reflect this situation. Thus for scheme *C* we get:

addr of stack administration → a_reg,
extend stack and save a_reg in s_reg,
values (calculated using a_reg) → stack block (using s_reg),
update stack administration (using s_reg).

We see that we need a special stack register s_reg for storing the values in the block and for the subsequential updating of the stack administration. The a_reg cannot serve, since it may be needed in calculating the values.

This code can be improved slightly by always having an empty block on top of the stack (the presence of which does not show in its max-limit):

addr of stack administration → s_reg,
values (calculated using a_reg) → stack block (using s_reg),
update administration (using s_reg),
extend stack (using s_reg).

Unfortunately the sequences for scheme *A* and scheme *C* have little in common. Moreover it would be nice not to deviate too much from the code for a **call**. We should like to use the existing **copies-to-input-gate** and **restores-from-output-gate**.

A certain measure of unification can be reached by the following sequence:

addr of stack administration → s_reg,	(1)
values → gate (copies to input gate),	(2)
extension part 1,	(3)
gate → stack block (restores from output gate),	(4)
extension part 2.	(5)

The exact meaning of each step under the various schemes is now easy to see, except for steps 2 and 4 under scheme *C*. Step 2 should store each value in exactly one location in the stack block; step 4 should spread them out, if necessary.

This determines the meaning of the **formals** in **copies-to-input-gate** and **extension-copies**:

- The first **integer** is the **position-on-gate**, i.e., the number of the **field-transport** in the ALEPH text.
- The second **integer** is the **position-on-stack**, i.e., the position in the stack block of one of the **selectors** in the **field-transport**.
  The gate is filled and unloaded stack-wise. The first **formal** in **restores-from-output-gate** is identical to the last one in **copies-to-input-gate**.

This means that the notion **extension-copies** is now obsolete and that the **extension-call** can rightfully obtain the symbol *EXC*.

Example: the translation of the **extension** (ALEPH Manual 3.4.3):

** 3 → ect, 5 → sel → ors * lst*

could be:

		$ scheme *A* :	scheme *C* :
step 2:	LVC 23,38	$ 3 ⇒ v_reg	3 ⇒ v_reg
	CVR 1,2	$ v_reg → g1	v_reg ⇒ w_reg → ect
	LVC 51,17	$ 5 ⇒ v_reg	5 ⇒ v_reg
	CVR 2,1	$ v_reg → g2	v_reg ⇒ w_reg → sel
step 4:	LDW 2,1	$ g2 ⇒ w_reg	sel ⇒ w_reg
	SWS 1	$ w_reg → sel	—
	SWS 3	$ w_reg → ors	w_reg → ors
	FRW		
	LDW 1,2	$ g1 ⇒ w_reg	ect ⇒ w_reg
	SWS 2	$ w_reg → ect	—
	FRW		

The above change also removes the last LL(1) conflict in the ALICE grammar (6.7).

6.6. A new ALICE instruction?

The practical use of ALEPH, especially for the implementation of finite-state machines [JONKERS 78], has led to the wish for an optimized translation of the dynamically last **affix-form** in an **actual-rule**. Under certain circumstances the resulting call can be implemented by abandoning the caller and replacing it by the called rule, i.e., by swapping.

The details are best understood when we examine the calling sequence of a call of S in R which has been called by Q, under the following assumptions:

- the call of S in R is dynamically the last call in R,
- R and S have the same number of formals,
- R and S have output formals in the same positions, where an 'output formal' is any **formal-variable** (ALEPH Manual 3.3.1) that ends in >,
- the **actual-affixes** in the call of S in the output positions are the corresponding **formal-affixes** of R.

These assumptions together form the 'swap condition'.

The dynamically last call of S in R which has been called by Q involves the following steps in scheme A (gate in registers):

Q R S	
some values → gate,	(1)
link to *S*,	(2)
allocate formals and locals of *S*,	(3)
gate → input formals of *S*,	(4)
do *S*,	(5)
output formals of *S* → gate,	(6)
deallocate formals and locals of *S*,	(7)
unlink,	(8)
gate → some locations,	(9)
output formals of *R* → gate,	(10)
deallocate formals and locals of *R*,	(11)
unlink,	(12)
gate → some locations.	(13)

Under the 'swap condition' the 'some locations' in (9) are exactly the 'output formals of *R*' in (10), with the consequence that (9) and (10) cancel out. Now the latest place where the formals and locals of *R* can still be used is (1). The sequence (11, 12) can therefore be moved upwards to after (1), where (12) coalesces with (2). We thus arrive at the following swap sequence under scheme *A*:

Q R&S	
some values → gate,	(1)
deallocate formals and locals of *R*,	(11)
unlink & link to *S*,	(2,12)
allocate formals and locals of *S*,	(3)
gate → input formals of *S*,	(4)
do *S*,	(5)
output formals of *S* → gate,	(6)
deallocate formals and locals of *S*,	(7)
unlink,	(8)
gate → some locations.	(13)

We see that what happens inside *S* (sequence (3-8)) has not changed, so *S* need never know it was called in an unusual way.

It should be noted that this optimization hinges on the fact that *S* cannot, by itself, access the formals of *R*. In ALEPH a rule can only access the globals and its own formals and locals. As a consequence, the corresponding optimization is not immediately valid in ALGOL 60 or ALGOL 68 (unless deeper analysis shows that there is no danger).

The optimization is less clear in scheme *C*:

Q R S

some values → actuals of R,	(1)
link to S,	(2)
allocate locals and actuals of S	
(actuals of R ⇒ formals of S),	(3)
do S,	(4)
deallocate locals and actuals of S	
(formals of S ⇒ actuals of R),	(5)
unlink,	(6)
actuals of R → some locations,	(7)
deallocate locals and actuals of R	
(formals of R ⇒ actuals of Q),	(8)
unlink,	(9)
actuals of Q → some locations.	(10)

Under the swap condition the 'some locations' in (7) are the 'formals of R'. We can again move (7,8) upwards, but only if we make the appropriate changes inside S:

Q R/S

some values → actuals of R,	(1)
actuals of R → formals of R,	(7)
deallocate locals and actuals of R	
(formals of R ⇒ actuals of Q),	(8)
unlink & link to S,	(2,9)
allocate locals and actuals of S	
(actuals of Q ⇒ formals of S),	(3Q)
do S,	(4)
deallocate locals and actuals of S	
(formals of S ⇒ actuals of Q),	(5Q)
unlink,	(6)
actuals of Q → some locations.	(10)

A problem lies in (3Q) and (5Q): S assumes the actuals of Q to be its (S's) formals, which is acceptable, provided that the number of formals of S be not greater than the number of actuals of Q. And indeed, because of the swap condition, the number of formals of S is equal to that of R, which in turn is less than or equal to the number of actuals of Q, so that no conflict can arise.

We are tempted to contract (1) and (7) into

some values → formals of R,	(1,7)

to make the sequence look more like a normal calling sequence. There is, however, a problem here: one of the values may be a formal of R, and since the transport in (1,7) is actually a sequence of transports cross-effects may occur, as in the following example:

```
FUNCTION gcd + >a + >b + c>:
    b = 0, a → c;
    less + a + b, gcd + b + a + c;    $ !!!
    divrem + a + b + ? + a, :gcd.
```

It seems unreasonable to require the compiler to check this: the loss from not

checking is small, and the properties resulting from the check are not easily formulated in terms of ALICE concepts.

If implementing a swap in scheme C is already more difficult than in scheme A, the idea breaks down completely in schemes $D1$ and $D2$. The rule Q, which is completely unaware of rule S, will never be able to take the formals of S for those of R. This has the interesting consequence that whatever form the swap-instruction in ALICE may take, the full calling sequence must still be provided. Thus the swap feature reduces to an add-on property of the calling sequence, and it is left to the ALICE processor to either implement or ignore the swap.

Thus we arrive at the following modification of ALICE. Both the **call** and **ext-call** macro sequence are supplied with an **unstack-and-swap**-option whose presence indicates that the above short-cut is allowed.

To keep the semantics of ALICE self-contained the semantics of the macro must be explained in ALICE terms. The presence of an **unstack-and-swap** in a **call** or **ext-call** means that:

- the true- and false-addresses of this **call** are the addresses of the success- and fail-tail of the rule;
- for each **restore-to-output-gate** in the **success-tail** of the called rule there is an identical **restore-to-output-gate** in the **success-tail** of the calling rule, and vice versa;
- for each such **restore-to-output-gate** with **position-on-gate** G and **position-on-stack** S there is a sequence

  loadw symbol, sp, G**,** S**, el,**
  storew stack var symbol, sp, S**, el,**
  free w_reg symbol, el

  in the **restores-from-output-gate** in this **call**.

6.7. The ALICE grammar

The above changes have resulted in the following grammar.

```
$ ALICE grammar: 820517.
$ An ALICE program is a sequence
$ of macros, comment lines, and
$ pragmat lines.

$ A macro has the form:

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

$ An ALICE-terminal-symbol is a
$ sequence of three letters.
$ A string is represented as an exact
$ copy of the ALEPH string,
$ including the surrounding quotes.

$ A comment line is a terminal
$ production of comment, which see.
$ It should be ignored.

$ A pragmat line is a terminal
$ production of pragmat, which see.
$ It may, in principle, occur between
$ any pair of macro lines. A portable
```

```
$ program should not contain any
$ pragmat lines.


$ ALICE-terminal-symbols with
$ their representations

$ macro-names:
add symbol;                          $ add
begin file adm symbol;               $ bfa
call id symbol;                      $ cll
call end symbol;                     $ cle
class box id symbol;                 $ cbi
class box end symbol;                $ cbe
class begin symbol;                  $ csb
class end symbol;                    $ cse
char denotation symbol;              $ chd
constant source symbol;              $ css
comment symbol;                      $ xxx
communication symbol;                $ cmm
copy address symbol;                 $ cad
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 index symbol;                    $ eix
end symbol;                          $ end
end values symbol;                   $ eva
exit symbol;                         $ ext
ext constant decl symbol;            $ ecd
ext fcall symbol;                    $ efc
ext scall symbol;                    $ esc
ext table length symbol;             $ etl
ext table decl symbol;               $ etd
ext call end symbol;                 $ ece
ext scall id symbol;                 $ esi
ext fcall id symbol;                 $ efi
extension id symbol;                 $ exi
extension start symbol;              $ exs
extension call symbol;               $ exc
extension end symbol;                $ exe
ext rule decl symbol;                $ erl
fail tail id symbol;                 $ fti
fallow symbol;                       $ flw
fcall symbol;                        $ fcl
free w_reg symbol;                   $ frw
index symbol;                        $ ind
input gate symbol;                   $ igt
int symbol;                          $ int
int fill symbol;                     $ itf
jump symbol;                         $ jmp
label symbol;                        $ lab
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
pragmat symbol;                      $ prg
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
unstack and return true symbol;      $ unt
unstack and return false symbol;     $ unf
unstack and swap symbol;             $ unw
variable symbol;                     $ var
zone bounds symbol;                  $ znb
zone value symbol;                   $ znv

$ delimiters:
space symbol;                        $ ' '
comma symbol;                        $ ,
end of line;        $ medium-dependent

$ parameters:
```

```
max int symbol;                $ mxi
min int symbol;                $ mni
int size symbol;               $ isz
word size symbol;              $ wsz
max char symbol;               $ mxc
max string length symbol;      $ msl
new line symbol;               $ nln
same line symbol;              $ sln
new page symbol;               $ npg
rest line symbol;              $ rln
numerical-tag symbol;          $ num
pointer-tag symbol;            $ ptr

comma-tag symbol;              $ com
space-tag symbol;              $ spc
min addr symbol;               $ mna
max addr symbol;               $ mxa

transport symbol;              $ trp
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
date symbol;                   $ dte
time symbol;                   $ tim
bool invert symbol;            $ biv
bool and symbol;               $ bnd
bool or symbol;                $ bor
bool xor symbol;               $ xor
left circ symbol;              $ lci
left clear symbol;             $ lcl
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
may be string pointer symbol;  $ myp
was symbol;                    $ was
next symbol;                   $ nxt
previous symbol;               $ prv
list length symbol;            $ lsl
unstack symbol;                $ utk
unstack to symbol;             $ ust
unqueue symbol;                $ unq
unqueue to symbol;             $ uqt
scratch symbol;                $ scr
delete symbol;                 $ del
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
back file symbol;              $ bkf

$ Other primitives used as parameters:
string;
    $ character sequence delimit-
    $ ed by quotes; quotes in the
    $ string are represented by
    $ quote-images ("")
character;
    $ except space and comma
```

integer.
    $ unsigned digit sequence

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

data:
    [constant sources],
    [ext constant decls],
    [variable decls],
    [list areas],
    [ext table decls],
    [list administrations],
    [file administrations].

rules:
    ext rule decls,
    rules and root.

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

status information:
    status symbol, sp,
    integer, co,
        $ maximum of all
        $ **size-of-input-gates** and
        $ **size-of-output-gates**
    integer, co,
        $ number of **values**
    integer, co,
        $ number of **variable-decls**
    integer, co,
        $ number of
        $ **file-administrations**
    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; sum of
        $ 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;
    ext 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;
    max string length 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;
    numerical-tag symbol;
    pointer-tag symbol.

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

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

ext table length:
    ext table length symbol, sp,
    location, co,
    string, el. $ the ALEPH string


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 builds. 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.


$ Data:
constant sources:
    constant source,
    [constant sources].

constant source:
    constant source symbol, sp,
    repr val pair, el.

repr val pair:
    repr, co, valref.

repr:
    integer.
        $ A repr either represents an
        $ ALICE object uniquely (>0)
        $ or it indicates the absence
        $ of an ALICE object (=0).


ext constant decls:
    ext constant decl,
        [ext constant decls].

ext constant decl:
    ext constant decl symbol, sp,
    repr, co, string, el.
        $ the ALEPH string


variable decls:
    variable decl, [variable decls].

variable decl:
    variable symbol, sp,
    repr val pair, co,
    repr, co, $ of next variable-decl
    string, el.
        $ the ALEPH tag in quotes


list areas:
    list area,
    [list areas].

list area:
    list symbol, sp,
    list area info, el,
    [list fillings],
    end list symbol, sp,
    list area info, el.
```

list area info:
repr, co, $ of the list
list type, co,
valref.
$ number of virtual addresses

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.**

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

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

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, $ actual min
valref, co, $ actual max
valref, co, $ calibre
repr, co, $ of next **list-info** or 0
string. $ the ALEPH tag in quotes

list type:
integer.
$ sum of:
$ 1: background pragmat
$ 2: breathing

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,
$ next **file-administration** or 0
string. $ the ALEPH string

file type:
integer.
$ sum of
$ 1: datafile
$ 2: input
$ 4: output

pointer area:
pointer symbol, sp,
repr, el, $ of a **list-info**
[pointer area].

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

communication area:
communication symbol, sp,
repr, co, $ first **list-info**
repr, co,
$ first **file-administration**
repr, co, $ first **variable-decl**

```
    string, el, $ ALEPH program title
    status information.


ext rule decls:
    ext rule decl, [ext rule decls].

ext rule decl:
    ext rule decl symbol, sp,
    repr, co, stag, el.

stag:
    string; $ the ALEPH 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:
    transport symbol;
    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;
    date symbol;
    time symbol;
    bool invert symbol;
    bool and symbol;
    bool or symbol;
    bool xor symbol;
    left circ symbol;
    left clear 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;
    may be string pointer symbol;
    was symbol;
    next symbol;
    previous symbol;
    list length symbol;
    unstack symbol;
    unstack to symbol;
    unqueue symbol;
    unqueue to symbol;
    scratch symbol;
    delete 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;
    back char symbol;
    back data symbol;
    back line symbol;
    back file symbol.


rules and root:
```

```
    [rule decls], root, [rule decls].

rule decls:
    rule decl, [rule decls].

root:
    root symbol, sp,
    integer, co,
        $ number of actuals of call
    string, el, $ program title
    source line,
    affix form,
    exit.


affix form:
    call;
    ext call.


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

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

rule id:
    rule id symbol, sp,
    rule triple, co,
    string, el.
        $ the ALEPH rule heading

rule triple:
    repr, co, rule type, co, recursion.

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

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

stack frame:
    stack frame symbol, sp,
    stack frame sizes, el.

stack frame sizes:
    integer, co, $ number of actuals
    integer, co, $ number of locals
    integer.
        $ maximum number of actuals in
        $ any call or ext-call in this rule


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 triple, 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 true symbol, sp,
    stack frame sizes, el.

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

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

unstack and return false:
    unstack and return false symbol, sp,
    stack frame sizes, el.
```

```
$ Gate handling in rules:
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 bodies:
rule body:
    statements.

statements:
    statement, [statements].

statement:
    call;
    ext call;
    primitive.

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

call:
    call id,
    [unstack and swap],
    input gate creation,
    [copies to input gate],
    scall or fcall,
    [restores from output gate],
    call end.

call id:
    call id symbol, sp, rule triple, el.

unstack and swap:
    unstack and swap symbol, sp,
    stack frame sizes, el.

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

size of input gate:
    integer.

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

call end:
    call end symbol, sp,
    true address, el.

false address:
    repr. $ of a label

true address:
    repr. $ of a label


ext call:
    ext call id,
    [unstack and swap],
    input gate creation,
    [copies to input gate],
    ext scall or ext fcall,
    [restores from output gate],
    ext call end.

ext call 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.

ext call end:
    ext call end symbol, sp,
    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,
    class box end symbol, sp,
    true address, el.
        $ the repr of a class


class:
    class begin symbol, sp,
    repr, el,
    zones,
    class end symbol, el.

zones:
    zone bounds, [zones];
    zone value, [zones].

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

minbound:
    repr val pair.

maxbound:
    repr val pair.

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


extension:
    extension id,
    input gate creation,
    copies to input gate,
    extension call,
    restores from output gate,
    extension end.

extension id:
    extension id symbol, el,
    load addr in a_reg, $ stack adm
    extension start symbol, el.

extension call:
    extension call symbol, el.

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


exit:
    exit symbol, sp,
    repr val pair, el.


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

```
$ Affix handling:
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:
    copy address symbol, el,
    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 val pair, 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:
    load index sequence,
    load list element in v_reg.

load index sequence:
    index symbol, el,
    load simple in v_reg,
    [load list element in v_reg sequence],
    end index symbol, el.

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
        $ 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 val to w_reg,
[store w_reg sequence],
free w_reg.

copy gate val to w_reg:
loadw symbol, sp, formal, el.

store w_reg sequence:
store w_reg, [store w_reg sequence].

store w_reg:
store w_reg in variable;
store w_reg in indexed element;
store w_reg in stack var.

store w_reg in variable:
storew variable symbol, sp, repr, el.

store w_reg in indexed element:
load index sequence,
store w_reg in list element.

store w_reg in list element:
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.

$ Miscellaneous:
pragmat:
pragmat symbol, sp,
string, co,
$ the ALEPH tag in quotes
integer, co,
$ 0: no **pragmat-value**
$ 1: **pragmat-value** was an **integer**
$ 2: **pragmat-value** was a **tag**
$ 3: **pragmat-value** was a **string**
string, el. $ the **pragmat-value**

comment:
comment symbol, sp,
string, el. $ to be ignored

7. REFERENCES

[AHO & ULLMAN 72] A.V. Aho & J.D. Ullman, The Theory of Parsing, Translation and Compiling, Vol. I, Prentice-Hall, 1972.

[AHO & ULLMAN 78] A.V. Aho & J.D. Ullman, Principles of Compiler Design, Addison Wesley Publ. Comp., 1978.

[BAUER & EICKEL 74] F.L. Bauer & J. Eickel (Eds.), Compiler Construction, An Advanced Course, Lecture Notes in Computer Science 21, Springer Verlag Berlin, 1974.

[BAYER et al. 81] M. Bayer et al., Software Development in the CDL2 Laboratory, *in* [HÜNKE 81].

[BÖHM 74] A.P.W. Böhm, Affixgrammatica's, afstudeerverslag (Affix Grammars, MSc. thesis), TH Delft, Delft, 1974, in Dutch.

[BÖHM 77] A.P.W. Böhm, ALICE: An Exercise in Program Portability, IW 91/77, Mathematical Centre, Amsterdam, 1977.

[BÖHM 78] A.P.W. Böhm, The Installation of ALICE on the PDP11/45 under UNIX, IW 94/78, Mathematical Centre, Amsterdam, 1978.

[BOSCH, GRUNE & MEERTENS 73] R. Bosch, D. Grune & L. Meertens, ALEPH, *A Language Encouraging Program Hierarchy*, *in* A. Günther et al. (Eds.), International Computing Symposium 1973, North-Holland Publ. Co., 1974; *also* IW 9/73, Mathematical Centre, Amsterdam, 1973.

[BOURNE, BIRRELL & WALKER 75] S.R. Bourne, A.D. Birrell & I.W. Walker, Z-code, an intermediate object code for ALGOL 68, The Computing Laboratory, Cambridge, 1975.

[BROWN 77] P.J. Brown (Ed.), Software Portability, An Advanced Course, Cambridge University Press, 1977.

[CATTELL 80] R.G.G. Cattell, Automatic Derivation of Code Generators from Machine Descriptions, ACM Trans. Program. Lang. Syst. *2*, 173-190, 1980.

[CLEAVELAND & UZGALIS 77] J. Craig Cleaveland & R.C. Uzgalis, Grammars for Programming Languages, Elsevier Scientific Publ. Co., Amsterdam, 1977.

[COMPASS 79] COMPASS Version 3 Reference Manual, #60492600, Control Data Corporation, Sunnyvale, Calf., 1979.

[CROWE 72] D. Crowe, Generating Parsers for Affix Grammars, Comm. ACM *15*, 728-734, 1972.

[CSIRMAZ 77] Csirmaz László, Az ALEPH programozási nyelv (The ALEPH Programming Language), I. és II. füzet, No. 17/1977, Mathematical Institute of the Hungarian Academy of Sciences, Budapest, 1977, in Hungarian.

[DAHL, DIJKSTRA & HOARE 72] O-J. Dahl, E.W. Dijkstra & C.A.R. Hoare,

Structured Programming, Academic Press, London, 1972.

[DEHOTTAY et al. 76] J.P. Dehottay, H. Feuerhahn, C.H.A. Koster & H.M. Stahl, Syntaktische Beschreibung von CDL2, Forschungsbericht Technische Universität Berlin, 1976, in German.

[DEREMER 71] F.L. DeRemer, Simple LR(k) grammars, Comm. ACM, *14*, 453-460, 1971.

[VAN DIJK 82] F. van Dijk, The Implementation of a Machine-Independent ALEPH Compiler, to be published.

[DIJKSTRA 75] E.W. Dijkstra, Guarded Commands, Nondeterminacy and Formal Derivation of Programs, Comm. ACM *18*, 453-457, 1975.

[DIJKSTRA 76] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, 1976.

[EARLEY & STURGIS 70] J. Earley & H. Sturgis, A Formalism for Translator Interactions, Comm. ACM *13*, 607-617, 1970.

[FEUERHAHN & KOSTER 78] H. Feuerhahn & C.H.A. Koster, Static Semantic Checks in an Open-ended Language, *in* P.G. Hibbard & S.A. Schuman (eds.), Constructing Quality Software, North-Holland Publ. Comp., 1978.

[FLORIJN & ROLF 81] G. Florijn & G. Rolf, PGEN — A General-Purpose Parser Generator, IW 157/81, Mathematical Centre, Amsterdam, 1981.

[FLOYD 63] R.W. Floyd, Syntactic Analysis and Operator Precedence, J. ACM *10*, 316-333, 1963.

[GLANDORF, GRUNE & VERHAGEN 78] R. Glandorf, D. Grune & J. Verhagen, A W-grammar of ALEPH, IW 100/78, Mathematical Centre, Amsterdam, 1978.

[GRUNE, MEERTENS & VAN VLIET 73] D. Grune, L.G.L.T. Meertens & J.C. van Vliet, Grammar-handling Tools Applied to ALGOL 68, IW 5/73, Mathematical Centre, Amsterdam, 1973.

[GRUNE, BOSCH & MEERTENS 74] D. Grune, R. Bosch & L.G.L.T. Meertens, ALEPH Manual, IW 17/74, Mathematical Centre, Amsterdam, 1974.

[GRUNE 75] D. Grune, ALEPH, een grammaticale aanpak van programmacorrectheid (ALEPH, A Grammatical Approach to Program Correctness), *in* J.W. de Bakker, Colloquium Programmacorrectheid, MC Syllabus 21, Mathematical Centre, Amsterdam, 1975, in Dutch.

[GRUNE 77] D. Grune, Choosing a Tag-list Algorithm for a Compiler, with Special Application to the ALEPH Compiler, Software — Practice & Experience *9*, 575-593, 1979; *also* IW 89/77, Mathematical Centre, Amsterdam, 1977.

[GRUNE 81] D. Grune, From VW-grammar to ALEPH, *in* J.W. de Bakker & J.C. van Vliet, Algorithmic Languages, Proceedings of the International Symposium on Algorithmic Languages, North-Holland Publ. Comp., Amsterdam, 1981; *also* IW 162/81, Mathematical Centre, Amsterdam, 1981.

[HILL 72] I.D. Hill, Wouldn't it be nice if we could write computer programs in ordinary English, or would it?, Computer Bull. *12*, 306-312, 1972.

[HOPCROFT & ULLMAN 79] J.E. Hopcroft & J.D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley Publ. Comp., 1979.

[HÜNKE 81] H. Hünke, Software Engineering Environments, North-Holland Publ. Comp., 1981.

[JOHNSON & LESK 78] S.C. Johnson & M.E. Lesk, Language Development Tools, Bell Systems Technical J., *57*, 2155-2175, 1978.

[JONKERS 78] H.B.M. Jonkers, A Finite State Lexical Analyzer for the Standard Hardware Representation of ALGOL 68, IW 98/78, Mathematical Centre, Amsterdam, 1978.

[KERNIGHAN & RITCHIE 78] B.W. Kernighan & D.M. Ritchie, The C Programming Language, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1978.

[KNUTH 68] D.E. Knuth, Semantics of Context-Free Languages, Math. Systems Theory *2*, 127-145, 1968.

[KNUTH 71] D.E. Knuth, Top-Down Syntax Analysis, Acta Informatica *1*, 79-110, 1971.

[KOK 77] G. Kok, Programmeerkursus in de taal ALEPH (Programming course for the language ALEPH), Report 25-77-207, Subfaculteit Psychologie, University of Amsterdam, Amsterdam, 1977, in Dutch.

[KOSTER 65] C.H.A. Koster, On the construction of ALGOL-procedures for generating, analysing and translating sentences in natural languages, MR 72, Mathematical Centre, Amsterdam, 1965.

[KOSTER 71a] C.H.A. Koster, A Compiler Compiler, MR 127/71, Mathematical Centre, Amsterdam, 1971.

[KOSTER 71b] C.H.A. Koster, Affix Grammars, *in* J.E.L. Peck (Ed.), ALGOL 68 Implementation, North-Holland Publ. Co., Amsterdam, 1971.

[KOSTER 72] C.H.A. Koster, Towards a Machine-Independent ALGOL 68 Translator, MR 129/72, Mathematical Centre, Amsterdam, 1971.

[KOSTER 74] C.H.A. Koster, Using the CDL compiler, *in* [BAUER 74].

[KÜHLING 78] P. Kühling, Affix-grammatiken zur Beschreibung von Programmiersprachen, Diss. D83, Technische Universität Berlin, 1978.

[LEVERETT et al. 80] B.W. Leverett et al., An Overview of the Production-Quality Compiler-Compiler Project, Computer *13*, 38-49, 1980.

[LEWIS II, ROSENKRANTZ & STEARNS 76] P.M. Lewis II, D.J. Rosenkrantz & R.E. Stearns, Compiler Design Theory, Addison-Wesley Publ. Comp., 1976.

[LINGER, MILLS & WITT 79] R.C. Linger, H.D. Mills & B.I. Witt, Structured

Programming: Theory and Practice, Addison-Wesley Publ. Comp., 1979.

[MARCOTTY & LEDGARD 76] M. Marcotty & H.F. Ledgard, A Sampler of Formal Definitions, Comp. Surveys *8*, 191-276, 1976.

[MEIJER 80] H. Meijer, An Implementation of Affix Grammars, *in* N.D. Jones (Ed.), Semantics-Directed Compiler Generation, Lecture Notes in Computer Science 94, Springer Verlag Berlin, 1980.

[NOS/BE 79] NOS/BE Version 1 Reference Manual, #60493800, Control Data Corporation, St. Paul, Minn., 1979.

[RICHARDS 77] M. Richards, Portable Compilers, *in* [BROWN 77].

[RITCHIE & THOMPSON 74] D.M. Ritchie & K. Thompson, The UNIX Time-sharing System, Comm. ACM *17*, 365-375, 1974.

[SHERIDAN 59] P.B. Sheridan, The Arithmetic Translator-Compiler of the IBM FORTRAN Automatic Coding System, Comm. ACM *2*, 9-21, 1959.

[SINTZOFF 67] M. Sintzoff, Existence of a Van Wijngaarden grammar for every recursively enumerable set, Annales de la Société Scientifique de Bruxelles *81*, 115-118, 1967.

[STAHL 78] H.-M. Stahl, Portability and Efficiency in Using an Open-Ended Language, Report #9, Informatica/Computer Graphics, Nijmegen University, The Netherlands, 1978.

[TANENBAUM, KLINT & BÖHM 77] A.S. Tanenbaum, P. Klint & W. Böhm, Guidelines for Software Portability, Software — Practice & Experience *8*, 681-698, 1978; *also* IW 88/77, Mathematical Centre, Amsterdam, 1977.

[WAITE 75] W.M. Waite, Implementing Software for Non-numeric Applications, Prentice-Hall Series in Automatic Computation, Prentice-Hall, 1975.

[WATT 77] D.A. Watt, The Parsing Problem for Affix Grammars, Acta Inf. *8*, 1-20, 1977.

[WICHMANN 77] B.A. Wichmann, How to Call Procedures, or Second Thoughts on Ackermann's Function, Software — Practice & Experience *7*, 317-329, 1977.

[VAN WIJNGAARDEN 65] A. van Wijngaarden, Orthogonal design and description of a formal language, MR 76, Mathematical Centre, Amsterdam, 1965.

[VAN WIJNGAARDEN 74] A. van Wijngaarden, The generative power of two-level grammars, *in* J. Loecks (Ed.), Automata, Languages and Programming, Lecture Notes in Computer Science 14, Springer Verlag Berlin, 1974.

[VAN WIJNGAARDEN 75] A. van Wijngaarden et al. (Eds.), Revised Report on the Algorithmic Language ALGOL 68, Acta Informatica *5*, 1-236, 1975; *also* MC Tract 50, Mathematical Centre, Amsterdam, 1976; *also* SIGPLAN Notices *12*, (5), 5-70, 1977.

[VAN WIJNGAARDEN 81] A. van Wijngaarden, Languageless Programming, *in* J. K.

Reid (Ed.), Proceedings of the IFIP/TC2/WG2.1 Working Conference on the Relations between Numerical Computation and Programming Languages, Boulder, North-Holland Publ. Comp., 1981; *also* IW 181/81, Mathematical Centre, Amsterdam, 1981.

[WILLIS 81] R.R. Willis, AIDES: Computer Aided Design of Software Systems II, *in* [HÜNKE 81].

[WULF & SHAW 73] W. Wulf & M. Shaw, Global Variable Considered Harmful, SIGPLAN Notices *8*, (2), 28-34, 1973.

[WULF et al. 75] W. Wulf et al., Design of an Optimizing Compiler, American Elsevier Publ. Comp., New York, 1975.

8. SUMMARY

This thesis reports on many aspects of the ALEPH project. The design and implementation of a programming language, even of a small one, requires work to be done on many subjects: semantics, syntax, lexical appearance, data structures, in- and output, parsing, error-recovery, run-time system, and portability, to mention a few. To master this complexity we need structure, and indeed part of the work went into structuring the rest of the work. This is reflected in this thesis, which deals partly with design and implementation *techniques* and partly with the design and implementation itself; this theme is expounded in 1.3, 4.2, 4.2.3, 4.4.2 and 4.4.3.

ALEPH is designed to foster good programming, to be simple and efficient, and to yield portable programs; its main target field is non-numeric programming (1.1). It is based on the analogy between formal grammars and programs [KOSTER 71a, GRUNE 75]. The frame-work of a two-level grammar (particularly of the variants 'VW-grammar' (2.3) and 'affix-grammar' (2.4)) is considered as a programming language (3.3.1). This language is then subjected to implementability requirements, in accordance with the design criteria (3.3.2). Of the ALEPH data structures, 'stacks' are of special interest (3.5.1). Some compromises in the design of the language are described in 3.6.

The portability of a program is endangered by a variety of problems [TANENBAUM, KLINT & BÖHM 77]. Many of them cannot materialize in an ALEPH program; the others are treated in 3.4. The greatest portability problem in a compiler is the machine-dependence of the code it generates (4.2.2); the ALEPH compiler avoids the problem by generating ALICE, a strictly machine-independent (ALEPH-related) intermediate code (4.4) [BÖHM 77].

The existence of a well-defined intermediate code has had a profound influence on the design of the compiler (chapter 4). The design of the compiler was factorized into two stages: first an inventory was made of all information needed by the ALICE code, and then ways were devised to extract this information from the ALEPH text. The structure of ALICE allows this information to be split up into groups in a natural way. This technique and its consequences are discussed in 4.2; part of the resulting design of the compiler is shown in chapter 5. F. van Dijk used the design to build, in ALEPH, a compiler from ALEPH to ALICE; a subsequent processor from ALICE to COMPASS implements ALEPH on the Control Data Cyber.

The resulting compiler reflects the structure of ALICE. Rather than scanning and adjusting the input several times until the desired code results, the compiler reads the ALEPH program one single time and distributes the information it finds over several streams, which correspond to the ALICE sections. These streams are then processed (in the order dictated by ALICE) into the ALICE translation of the ALEPH program (4.3.1).

The use of ALICE as a strict target interface in the design of the compiler put higher demands on ALICE than it could meet. The techniques used in the design of ALICE were analysed and sharpened into the 'parallel-script' technique (4.4.3) which was then used to improve ALICE (chapter 6).

9. SAMENVATTING

In dit proefschrift wordt ingegaan op een aantal facetten van het ALEPH project. Bij het ontwerpen en implementeren van een programmeertaal, hoe klein ook, moet aandacht worden besteed aan een veelheid van onderwerpen, als daar zijn: semantiek, syntax en lexicaal aanzien van de taal, de data-structuren, de in- en uitvoer, de syntactische analyse (met foutherstelling), het run-time systeem en de overdraagbaarheid, zowel van de vertaler als van de in de taal geschreven programma's. Willen we temidden van al deze taken het overzicht blijven houden, dan moeten we structurerend optreden; een gedeelte van de tijd, die besteed is aan het ALEPH project, is dan ook gaan zitten in het structureren van de taken. Dit verschijnsel vindt zijn weerslag in dit proefschrift: naast het ontwerp van de taal en dat van de vertaler worden ook de technieken die tot deze ontwerpen geleid hebben, uitgebreid behandeld (1.3, 4.2, 4.2.3, 4.4.2 en 4.4.3).

Het ALEPH project had tot doel een eenvoudige en efficiënte taal te ontwerpen, die bevorderlijk is voor een goede programmeerstijl; de programma's moesten gemakkelijk overdraagbaar zijn; als toepassingsgebied werd voornamelijk aan niet-numerieke programmatuur gedacht (1.1). De taal is gebaseerd op de analogie tussen formele grammatica's en programma's [KOSTER 71a, GRUNE 75]. Het formalisme van een twee-niveau grammatica (in het bijzonder dat van twee varianten, 'vW-grammatica's' (2.3) en 'affix-grammatica's' (2.4)) wordt beschouwd als een programmeertaal (3.3.1). Deze taal wordt vervolgens onderworpen aan implementatie-eisen die voortvloeien uit de ontwerpcriteria (3.3.2). Van de ALEPH data-structuren zijn vooral de 'stacks' van belang (3.5.1). Enige compromissen in het ontwerp van de taal worden behandeld in 3.6.

Bij het overdragen van een programma moet in het algemeen met een reeks problemen rekening worden gehouden [TANENBAUM, KLINT & BÖHM 77]. Vele van deze problemen kunnen zich echter in ALEPH niet voordoen; de overige worden behandeld in 3.4. Bij een vertaler doet zich een specifiek overdraagbaarheidsprobleem voor, namelijk de machine-afhankelijkheid van het resultaat (4.2.2); de ALEPH vertaler omzeilt het probleem door een vertaling te genereren in een zuiver machine-onafhankelijke, op ALEPH toegespitste tussencode, ALICE (4.4) [BÖHM 77].

Het bestaan van zo'n scherp gedefinieerde tussencode heeft grote invloed gehad op het ontwerpproces van de vertaler (hoofdstuk 4). Dit proces kon in twee fasen worden gesplitst: eerst werd een inventaris opgemaakt van alle gegevens benodigd voor de ALICE code, en vervolgens werden er manieren ontwikkeld om deze gegevens aan de ALEPH tekst te ontlokken. De structuur van ALICE is zodanig dat deze gegevens op natuurlijke wijze in groepen ingedeeld kunnen worden. De gebruikte techniek wordt besproken in 4.2; een gedeelte van het hieruit voortvloeiende uiteindelijke ontwerp van de vertaler staat beschreven in hoofdstuk 5.

Uitgaande van dit ontwerp heeft Frank van Dijk een vertaler van ALEPH naar ALICE geschreven in ALEPH; met behulp van een omvormer van ALICE naar COMPASS wordt een implementatie op de CDC Cyber gerealiseerd.

De aldus verkregen vertaler weerspiegelt de structuur van ALICE. Terwijl een klassieke vertaler de invoertekst ettelijke malen leest en bijschaaft, tot een bevredigende vertaling ontstaan is, leest de huidige vertaler de ALEPH-invoertekst éénmaal en verdeelt de gevonden gegevens over een aantal informatiestromen, elk overeenkomend met een bepaald ALICE onderdeel. Deze stromen worden dan, in een door de structuur van ALICE voorgeschreven volgorde, verder verwerkt tot de ALICE-

vertaling van de ALEPH-invoertekst (4.3.1).

Het gebruik van ALICE als doeltaal en leidraad bij het ontwerp van de vertaler bleek hogere eisen aan ALICE te stellen dan waaraan de oorspronkelijke versie kon voldoen. De technieken, volgens welke ALICE ontworpen was, werden onderzocht en verscherpt; dit resulteerde in de 'parallel-script-techniek' (4.4.3), met behulp waarvan vervolgens een verbeterde versie van ALICE ontworpen werd (hoofdstuk 6).

10. CURRICULUM VITAE

Naam:	Grune, Dick.
Geboren:	7 december 1939, te Enschede.
1958:	Eindexamen Gymnasium β, Gemeeentelijk Lyceum te Enschede.
1962:	Kandidaatsexamen Natuur- en Scheikunde, Rijksuniversiteit Utrecht.
1967:	Doctoraalexamen Experimentele Natuurkunde, Rijksuniversiteit Utrecht.
1967:	Gehuwd met Lily Ossendrijver.
1968-1970:	Systeemprogrammeur, Technion, Haifa, Israel.
1970-heden:	Wetenschappelijk Medewerker, Mathematisch Centrum, Amsterdam.

11. INDEX

Notions from the ALICE grammar can be found in 6.7; they are not included in this index.

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

ALEPH MANUAL

R. Bosch
D. Grune
L.G.L.T. Meertens

Fourth printing

Mathematical Centre

1982

0. PREFACE

ALEPH (acronym for '*A Language Encouraging Program Hierarchy*') is a high-level language designed to provide the programmer with a tool that will effectively aid him in structuring his program in a hierarchical fashion. The syntactic and semantic simplicity of ALEPH leads to efficient object code [WICHMANN 77], so that the loss of efficiency usually incurred in structured programming is avoided. ALEPH is suitable for any problem that suggests top-down analysis (parsers, search algorithms, combinatorial problems, artificial intelligence problems, etc.).

Chapter one of this Manual gives a tutorial introduction into the way of thinking who is used in ALEPH. It addresses itself to computer users that have some experience with algorithms and grammars. It must not be concluded from these prerequisites that ALEPH should not be taught to the novice programmer. On the contrary, ALEPH introduces him to a discipline of thought that is lacking in many other languages.

Chapter two treats the ALEPH program in general terms. Chapter three through six contain a complete description of ALEPH.

Chapter three treats the flow-of-control. Chapter four treats the data-types. Externals, i.e., standard-operations and communication with the outside world, are treated in chapter five. Chapter six describes the pragmats.

The representations of 'symbols' and example programs are given in chapters seven and eight.

An ALEPH compiler exists, which translates ALEPH programs into ALICE programs in a machine-independent fashion. ALICE [BÖHM 77] is a simple linear code designed to aid the installation of ALEPH on new systems. The ALEPH compiler is available in both ALEPH and ALICE. An ALICE transformer to COMPASS for the Cyber 170 is also available. The *compile-* and *count*-pragmats have not been implemented.

This is the fourth printing of the ALEPH Manual. Many paragraphs have been rephrased to remove inconsistencies; the paragraph numbers have been kept identical throughout all printings. Since the third printing in 1977 the following modifications have been made.

3.4.3: the **source**s in an **extension** are evaluated *before* the stack is extended. This prevents the **extension** $* \; st[>>st] \rightarrow st * st$ from pushing uninitialized data on the stack *st*.

3.5: no match in calibre is required between a formal and an actual list if the former is explicitly declared with zero selectors, rather than with one selector. This criterion is clearer and prevents the misapplication of some standard externals.

5.2.5: the standard externals *back char*, *back data* and *back line* are deleted since their limited usefulness in no way justifies the effort needed in their implementation.

6.1: to be effective a *macro*-pragmat must occur *before* the pertinent **rule-declaration**. This modification greatly increases the efficiency of the translation process.

6.1, 6.2: all pragmats to switch off run-time checking have been deleted.

1. AN INFORMAL INTRODUCTION TO ALEPH

In this chapter we shall gradually develop a small ALEPH program and intersperse it liberally with annotations and arguments. This introduction is intended to give some insight into the use of the language ALEPH and to display its main features in a very informal way.

1.1. A grammar

The problem we shall treat is the following. We want to write a program that reads a series of arithmetic expressions separated by commas, calculates the value of each expression while reading it, and subsequently prints the value. The expressions will contain only integers, plus-symbols, times-symbols and parentheses: an example might be '15×(12+3×9)'.

First we put the requirements for the input to our program in the more transparent and clearer form of a context-free grammar. This grammar shows exactly which symbol we will accept in which position.

input: expression, input tail.
input tail: comma symbol, input; empty.
expression: term, plus symbol, expression; term.
term: primary, times symbol, term; primary.
primary: left parenthesis, expression, right parenthesis; integer.
integer: digit, integer; digit.
empty: .

The rule for **input** can be read as: **input** is an **expression** followed by an **input-tail**, whereas the rule for **primary** can be read as: a **primary** is either

- a **left-parenthesis** followed by an **expression** followed by a **right-parenthesis**, or
- an **integer**.

This grammar shows clearly that for instance '15× +3' will not be accepted as an **expression**. The '×' can only be followed by a **term**, which always starts with a **primary**, which in turn either starts with an **integer** or a **left-parenthesis**, but never with a '+'.

1.2. Rules

We shall now write a collection of rules in ALEPH, one for each rule in the grammar. For the grammar rule for **expression** we shall write an ALEPH rule that, when executed, reads and processes an expression and yields its result. This ALEPH rule looks as follows:

```
ACTION expression + res> - r:
    term + res,
        (is symbol + /+/, expression + r, plus + res + r + res;
        +).
```

This can be read as: an **expression**, which must yield a result in *res* and uses a (local) variable *r* is (we are now at the colon) a **term** which will yield a result in *res*, followed *either* (we are now at the left parenthesis) by a **plus-symbol** followed by an **expression** which will yield its result in *r* after which the result in *res* and the result in *r* will be added to form a new result in *res*, *or* (we are at the semicolon now) by

nothing. We see that this is the old meaning of the grammar rule for **expression**, sprinkled with some data-handling. The data-handling tells what is to be done to get the correct result: we could call it the semantics of an **expression**. If we remove these paraphernalia from the ALEPH rule we obtain something very similar to the original grammar rule:

ACTION expression 1:
term, (is symbol + /+/, expression 1; +).

This rule, while still correct ALEPH, does no data handling and, consequently, will not yield a result; it could for example be used to skip an **expression** in the input.

We now direct our attention back to the ALEPH rule *expression* and consider what happens when it is 'executed'. First, *term* is executed and will yield a result in *res*: it does so because we shall define *term* so that it will. Then we meet a series of two alternatives separated by a semicolon (*either* a this *or* a that). First an attempt is made to execute the first alternative by asking *is symbol + /+/*. This is a question (because we shall define it so) which is answered positively if indeed the next symbol is a '+' (in which case the '+' will be discarded after reading) or negatively if the next symbol is something else.

If *is symbol + /+/* 'succeeds' the remainder of the first alternative is executed, *expression + r* is called (recursively), yielding its result in *r* and subsequently *plus + res + r + res* is called, putting the sum of *res* and *r* in *res*. The call of *expression + r* works because we just defined what it should do. *plus* is a name known to the compiler and has a predefined meaning. However, if we are dissatisfied with its workings we could define our own rule for it. Now this alternative is finished, so the parenthesized part is finished, which brings us to the end of the execution of the rule *expression*.

If *is symbol + /+/* 'fails' the second alternative is tried: the part after the semicolon. This alternative consists of a + which is a dummy statement that always succeeds. Without further action we reach the end of the rule *expression*.

The above indicates the division of responsibilities between the language and the user. The language provides a framework that controls which rules will be called depending on the answers obtained from other rules. The user must fill in this framework, by defining what actions must be performed by a specific rule and what questions must be asked. These definitions will again have the form of rules that do something (to be defined by the user) embedded in a framework that controls their order (supplied by the language). It is clear that this process must end somewhere. It can end in one of two ways.

It may turn out that the action needed is supplied by ALEPH: there are three basic primitives in the language, the copying of a value, the test for equality of two values and the extension of a stack by a fixed number of given values. Often, however, these three primitives are not sufficient to express the action needed; the rule is then subdivided into other rules. There are, however, cases where this is not desirable (or not possible). In such cases the rule is declared 'external' and its actions must be specified in a different way, often in the assembly language of the machine used. By specifying a rule as 'external' we leave the realm of machine-independent semantics. A number of external rules are predefined by the compiler, including the rule *plus* used above. This set of rules will suffice for most applications.

We shall now pay some attention to the exact notation (syntax) of the rule *expression*. All rules have the property that when they are called they are either guaranteed to succeed or they may fail. The word *ACTION* indicates that a call of this rule is guaranteed to succeed. The name of the rule is *expression* and *res* is its only formal 'affix' (parameter). The + serves as a separator (it 'affixes' the affix to the rule). The right arrow-head (>) indicates that the resulting value of *res* will be passed back to the calling rule. This means that *expression* has the obligation to assign a value to *res* under all circumstances: *res* is an output parameter, guaranteed to receive a value. If the text of the rule does not support this claim, the compiler will discover thus and issue a message. The +-sign and the term 'affix' stem from the theory of affix grammars on which ALEPH is based [KOSTER 71b, WATT 77].

The $-r$ specifies r as a local affix (local variable) of the rule and the colon closes the left hand side. The + in *term + res* appends the actual affix *res* to the rule *term*, the comma separates calls of rules. The parentheses group both alternatives into one action. The + between slashes (indicating 'absolute value') represents the integer value of the plus-symbol in the code used. The semicolon separates alternatives, which are checked in textual order. As said before, the stand-alone + denotes the dummy action that always succeeds. The period ends the rule.

The following approximate translation to ALGOL 68 may be helpful:

```
PROC expression = (REF INT res) VOID:
BEGIN INT r;
    term(res);
    IF is symbol("+")
    THEN expression(r); plus(res, r, res)
    ELSE SKIP
    FI
END
```

1.3. Further rules

In view of the above the rule for *term* should not surprise the reader:

```
ACTION term + res> - r:
    primary + res,
        (is symbol + /×/, term + r, times + res + r + res; +).
```

Now we are tempted to render the rule for *primary* as:

```
ACTION primary + res>:
    is symbol + /(/, expression + res, is symbol + /)/;
    integer + res.
```

but here the compiler would discover that we did not specify what should be done if the second call of *is symbol* fails. If that happens, we would have recognized, processed and skipped a **left-parenthesis** and a complete **expression**, to find that the corresponding **right-parenthesis** is missing. This means that the input (which is a production of **input**) is incorrect; we now decide that we shall not do any error recovery, so we give an error message and stop the program. The correct version of the ALEPH rule *primary* is then:

```
ACTION primary + res>;
    is symbol + /(/, expression + res,
        (   is symbol + /)/;
            error + no paren
        );
    integer + res.
```

Here the two alternatives between parentheses behave like one action that will always succeed: either the right parenthesis is present in the input, or an error will be signalled. *no paren* is a constant that will be specified later on.

Writing the rule for *integer* is a trickier problem than it seems to be. For a comprehensive account on how to obtain correct and incorrect versions the reader is referred to [KOSTER 71a]. We shall confine ourselves to giving one correct version. It consists of two rules and is about as complicated as is necessary.

```
ACTION integer + res>:
    digit + res, integer 1 + res;
    error + no int, 0 → res.

ACTION integer 1 + >res> − d:
    digit + d, times + res + 10 + res, plus + res + d + res, integer 1 + res;
    +.
```

The rule *integer* asks for a digit. If present, its value will serve as the initial value of *res*. The value of *res* is then passed to *integer 1*. If no digit is present an error message will result and *res* will get the dummy value *0*. This is necessary to ensure that *integer* will assign a value to *res* under all circumstances (because of the right arrow-head after *res*). The right arrow in *0* → *res* designates the assignation of the value on the left to the variable on the right, one of the primitive actions in ALEPH.

The rule *integer 1* processes the tail of the integer. If there is such a tail it starts with a digit, so the first alternative asks *digit + d*. If so, a new result is calculated from the previous one and the digit *d* by making *res* equal to *res* × *10* + *d* and *integer 1* is called again (to see if there are more digits to come). If there was no digit, we will have processed the whole integer and *res* contains its value.

The right arrow-head in front of *res* means that the calling rule will have assigned a value to this formal affix just before calling *integer 1*, i.e. *res* is 'initialized'. The right arrow-head after *res* again indicates that the resulting value will be passed back to the calling rule.

A more convenient way of reading an integer is provided by the (standard) external rule *get int*.

1.4. Input

The above forms the heart of our program. We shall now supply it with some input and output definitions. For the input we need a file to obtain the input symbols from, which we shall call *reader*; let us suppose that this file is called "SYSIN" somewhere in the surrounding operating system (e.g. on a control card). Furthermore we shall use a global variable *buff* which will contain the first symbol not yet recognized. Comment starts with a *$*.

```
$ Input
CHARFILE reader = >"SYSIN".
VARIABLE buff = / /.
```

The variable *buff* is initialized with the code for the space symbol (there being no uninitialized variables in ALEPH). We are now in a position to give two rule definitions that were still missing.

```
PREDICATE is symbol + >n: buff = n, get next symbol.

PREDICATE digit + d>:
    =buff=
    [/0/ : /9/], minus + buff + /0/ + d, get next symbol;
    [ : ], -.
```

These require some further explanation, mainly concerning the notation. The word *PREDICATE* indicates that *is symbol* is not an action but a question, or more precisely a 'committing' question as opposed to a 'non-committal' question. A non-committal question is a question that, regardless of the answer it yields, makes no global changes, does not do anything irreversible. A committing question is a question that, when answered positively, does make global (and often irreversible) changes, as specified by the programmer. To give an example, 'Are there plane tickets for New York for less than $ 100?' is a non-committal question, whereas 'Are there plane tickets for New York for less than $ 100? If so, I want one' is a committing question.

In the case of *is symbol* the (committing) question is: 'Is the symbol in *buff* equal to the one I want? If so, advance the input and put the next symbol in *buff*.' The form *buff* = *n* is a test for equality and is one of the primitive operations in ALEPH. *get next symbol* will be defined below.

Again the right arrow-head in front of the formal affix *n* indicates that the calling rule will have assigned a value to it; the absence of a right arrow-head to the right of the *n* indicates that the value of *n* (which may have been changed!) will not be passed back to the calling rule.

The rule for *digit* (again a 'predicate') shows another feature of ALEPH, the 'classification'. For certain classes of values of *buff* one alternative will be chosen, for other classes a different alternative will be chosen. The classes are presented inside the square brackets. Thus, for values of *buff* that lie between the code for '0' and the code for '9' the first alternative will be chosen. For all other values the dummy question that always fails (−) will be executed. The rule *digit* is equivalent to

```
PREDICATE digit 1 + d>:
    between + /0/ + buff + /9/, minus + buff + /0/ + d, get next symbol.
```

assuming that *between* + /0/ + *buff* + /9/ succeeds if and only if /0/ ⩽ *buff* ⩽ /9/. In complicated cases a classification is easier to write and will in general produce more efficient object code. The classification is analogous to case statements in ALGOL 68 and other programming languages.

All the arithmetic used here on symbols is based on the (possibly machine-dependent) assumption that the numerical codes associated with the symbols '0' through '9' are a set of consecutive integers in ascending order. The numerical value of a digit symbol can then indeed be obtained by subtracting the code for '0' from its

numerical value.

One more input rule must be supplied:

```
ACTION get next symbol:
    get char + reader + buff,
        (   (buff = / /; buff = new line), get next symbol;
            +
        );
    stop → buff.

CONSTANT stop = −1.
```

get char is an (external) rule known to the compiler. It tries to read the next symbol from the file identified by its first formal affix (here *reader*). If there is a symbol it puts it in its second formal affix (here *buff*); if there is no symbol it fails. In the latter case *buff* is given the value *stop*, which is defined in a 'constant declaration' to be -1.

If *get char* does yield a symbol and if it is a space or a new-line, *get char* is called again. We use nested parenthesizing here. This definition of *get char* implies that we have decided that spaces and new-lines are allowed in the input in all positions (a decision that was not yet present in the initial grammar).

1.5. Output

The output is as follows:

```
$ Output
CHARFILE printer = "SYSOUT">.

ACTION print integer + >int:
    out integer + int, put char + printer + new line.

ACTION out integer + >int − rem:
    divrem + int + 10 + int + rem, plus + rem + /0/ + rem,
        (int = 0; out integer + int),
        put char + printer + rem.
```

The rule *put char* is known to the compiler, as is *divrem*. The call of the latter has the effect that *int* is divided by 10, the quotient is placed back in *int* and the remainder in *rem*. This splits the number into its last digit and its head; if this head (now in *int*) is not zero it must be printed first, which is effected by the recursive call of *out integer*. Subsequently, the last digit is printed through a call of *put char*. This is a simple but inefficient way of printing a number. A more convenient way of printing an integer is provided by the (standard) external rule *put int*.

For the printing of error-messages we shall need some string handling. Strings do not constitute a special data type in ALEPH: they are handled, like all other complicated data types, by putting them in 'stacks' and 'tables' and are operated upon by suitably defined rules (generally defined by the programmer but sometimes predefined in the system).

The error handler takes the following form:

```
$ Error-message printing
ACTION error + >er:
    put char + printer + new line,
        put string + printer + strings + er, EXIT 1.

TABLE strings =
    (   "Right parenthesis missing": no paren,
        "Integer missing": no int
    ).
```

The table *strings* contains two strings, stored and packed in a way suitable to our machine; they can be reached under the names *no paren* and *no int*. The call of *put string* takes the formal affix *er*, looks in the table *strings* under the entry corresponding to *er* and transfers the string thus found to the file identified as *printer*.

When the construction *EXIT 1* is executed the program will be terminated and the 1 will be passed to the operating system as an indication of what went wrong. This is by no means the normal program termination: normal program termination ensues when all work is done.

1.6. Starting the program

The rule for reading an **expression** (*expression*) and the one for printing an **integer** (*print integer*) can now be combined into the rule *input* (see the grammar at the beginning of this chapter).

```
ACTION input - int:
    expression + int, print integer + int,
        (is symbol + /,/, input; +).
```

This rule combines the rules for **input** and **input-tail**. Instead of translating **empty** by + we could make a test to see whether we have indeed reached the end of the file:

```
(buff = stop; error + no end)
```

We now remember our convention that *buff* contains the first symbol not yet recognized, and realize that *buff* must be initialized with the first non-space symbol of the input:

```
ACTION initialize: get next symbol.

ACTION read expressions and print results: initialize, input.
```

The reader will have noticed that until now we have only defined rules that will do something if they are executed (called) and which will then call other rules. He may have wondered whether ALEPH contains any directly executable statements at all. The answer is yes, but only one (per program). In our example it has the following form:

```
ROOT read expressions and print results.
```

We now indicate the end of our program:

```
END
```

When the program is run the rule *read expressions and print results* is executed. This rule calls *initialize*, which through a call of *get next symbol* puts the first non-space symbol in *buff*; when *initialize* is done, *input* is called which calls *expression* which in turn executes *term*, etc. After a while *input*, which is called repeatedly, will find *is symbol + /,/* to fail, it is done, and so is *read expressions and print result*. The call specified in the *ROOT* instruction is finished: this constitutes the normal program termination.

We could give the 'rule declarations' and 'data declarations' in any other order and the effect would still be the same. The *END*, however, must be the last item of the program.

This brings us to the end of our sample program.

1.7. Some details

Although the rule *put string* used above is known to the compiler, it is useful to see, as an additional example, how it looks when expressed in ALEPH. We first propose the preliminary version *put string 1*.

```
ACTION put string 1 + ""file + table[] + >string - count:
    0 → count, next 1 + file + table + string + count.

ACTION next 1 + ""out + tbl[] + >str + >cnt - symb:
    string elem + tbl + str + cnt + symb, put char + out + symb,
        incr + cnt, next 1 + out + tbl + str + cnt;
    +.
```

The double set of quotation marks (*""*) indicates that the corresponding actual affix will be a file, the square brackets indicate that the corresponding actual affix will be a table. We see that the only thing *put string 1* does is to create an environment for *next 1* to run in. *next 1* starts by calling *string elem*. This (standard) rule considers the string in *tbl* designated by *str* and determines whether this string has a *cnt*-th symbol. If so, it puts it in *symb*; if not, it fails. If the call fails, we know we have reached the end of the string and we are done. Otherwise the symbol is transferred to the file identified by *out*, the counter *cnt* is increased by 1 (through the external rule *incr*) and *next 1* is called again with the same affixes. Like at the first call of *next 1*, the value of *cnt* is the position in the string of the symbol to be processed.

The recursive call of *next 1* is a case of trivial right-recursion; moreover all actual affixes are the same as the formal affixes (which are left of the colon). In this case the recursive call is equivalent to a straightforward jump: it does not even necessitate parameter transfers. For this case there is a shorthand notation: a name of a rule preceded by a colon denotes the re-execution of that rule with the affixes it had upon its initial call (of course this is only allowed inside that same rule and only if the recursion is trivial right-recursion). Now we can write a simplified version:

```
ACTION put string 2 + ""file + table[] + >string - count:
    0 → count, next 2 + file + table + string + count.
```

```
ACTION next 2 + ""out + tbl[] + >str + >cnt − symb:
    string elem + tbl + str + cnt + symb,
        put char + out + symb, incr + cnt, : next 2;
    +.
```

The gain is twofold. We no longer have to write a tail of affixes which only convey the information 'same as before', and, more important, the rule *next 2* is now called only in one place (in *put string 2*). This means that we could as well explicitly have written it there. We now replace the call of *next 2* in *put text 2* by the definition of *next 2*: we parenthesize the rule, substitute for each formal affix its corresponding actual affix and remove the formal affixes:

```
ACTION put string + ""file + table[] + >string − count:
    0 → count,
        (next − symb:
            string elem + table + string + count + symb,
                put char + file + symb, incr + count, :next;
            +
        ).
```

Note that this mechanism of replacing a call of a rule by its (slightly modified) definition is not applied here for the first time. We have been using it tacitly from the very first sample rule in 1.2. There the rule *expression* is a contraction of:

```
ACTION expression 1 + res>:
    term + res, expression tail 1 + res.
```

and

```
ACTION expression tail 1 + >res> − r:
    is symbol + /+/, expression 1 + r, plus + res + r + res;
    +.
```

which, according to the above recipe, would yield:

```
ACTION expression 2 + res> :
    term + res,
        (expression tail 2 − r:
            is symbol + /+/, expression 2 + r, plus + res + r + res;
            +
        ).
```

In a sense this is a more appropriate form than the one given in 1.2: now the *r* occurs where it belongs, that is, in the position of a local affix of the parenthesized part only. To obtain the exact version in 1.2 one must start from:

```
ACTION expression 3 + res> − r:
    term + res, expression tail 3 + res + r.
```

and

```
ACTION expression tail 3 + >res> + r>:
    is symbol + /+/, expression 3 + r, plus + res + r + res; +.
```

2. INTRODUCTION TO THE MANUAL

2.1. Interface with the outside world

The solution of a problem by means of a computer implies that a sequence of actions be specified that, when executed, lead to the desired result. In ALEPH the actions in this sequence may be obtained from four sources:

a. the framework of the language (supplied by the compiler),
b. the program (supplied by the programmer),
c. the standard externals (standard definitions of actions, to be supplied by the compiler if the need arises),
d. the programmer-defined externals (definitions of actions supplied by the programmer but not belonging to the program; for example, precompiled code or machine code).

The framework of ALEPH is treated in chapter 3, the program is treated in section 3.1 and the externals are treated in chapter 5.

The data needed in solving the problem at hand come from four sources:

a. the data descriptions in the program,
b. the input file(s),
c. the predefined constants in the compiler (e.g., the maximum value an integer can have),
d. the programmer-defined external values (in the rare case that these values cannot be normally defined in the program, as for example computer-generated binary tables of considerable size).

The data descriptions and the input files are explained in chapter 4, and the externals in chapter 5.

The results can be passed back to the outside world along two paths:

a. as output files,
b. as a single integer (the termination state of the program) which is made available to the operating system upon termination of the program, indicating in some way the outcome of the program.

The output files are described in section 4.2. The termination state is described in 3.1 and in 3.6. In some operating systems it can be used to control the further course of events, in other operating systems it may only indicate whether the program proceeded satisfactorily or broke off because of some irrecoverable error.

2.2. The syntactical description

The syntax of ALEPH is given in the form of a context-free grammar. The notation in this grammar follows a well-known scheme: the part on the right hand side of a syntax rule defines the possible productions of the notion on the left hand side. The right hand side consists of one or more alternatives, separated by semicolons, of which only one alternative applies in a given case. Sometimes one or more notions in an alternative are enclosed in square brackets: this indicates that the given notions may or may not be present, i.e., they are optional.

The terminal symbols of the grammar, together with their representations, are listed in 7.2; all except four end in **-symbol**. A notion that ends in **-tag** produces **tag**. Such a notion then contains a hint as to exactly which **tag**s are allowed by the context conditions. A full VW-grammar incorporating all context conditions was prepared by

R. Glandorf, D. Grune and J. Verhagen [GLANDORF, GRUNE & VERHAGEN 78].

Constituents of the grammar are printed in **bold**; programs and program fragments are printed in *script*.

3. PROGRAM LOGIC

3.1. General

3.1.1. The program

Syntax:

program:
    [information sequence], root, [information sequence], end symbol.
information sequence:
    information, [information sequence].
information:
    declaration; pragmat.
root:
    root symbol, affix form, point symbol.
declaration:
    rule declaration;
    data declaration;
    external declaration.

The syntax of **program** can be verbalized as: 'A **program** is a sequence of **declarations** and **pragmats**, followed by an **end-symbol**; in this sequence exactly one **root** must occur.' The order in which the **declarations** and the **root** appear is immaterial. The position of some **pragmats** is significant (6.1).

Example of a **program**:

```
CHARFILE output = "PRINTER">.
ROOT put char + output + /3/.
END
```

in which the first line is a **data-declaration**, the second is the **root** and the third contains the **end-symbol**. For other examples see chapter 8.

The execution of a **program** starts with the processing of all of its **data-declarations**, in such order that no data item is used before its value has been calculated. If no such order exists an error-message is given.

Example: the **data-declarations**

```
CONSTANT p = q.
CONSTANT q = 3.
```

are processed in reverse order, whereas the **data-declarations**

```
CONSTANT p = q.
CONSTANT q = 2 - p.
```

will result in an error-message.

A large part of the processing of the data-declarations will normally be performed during compilation.

After all constants, variables, stacks, tables and files have thus been established, the **affix-form** in the **root** is executed (3.5) as the sole directly executable instruction in the program. If this **affix-form** reaches its normal completion, the program finishes with a termination state of 0. If the execution of the **affix-form** stops prematurely, the program finishes, but now with a termination state possibly different from 0. If the stop is due to an **exit** instruction (3.6), the termination state is specified by this instruction. If the stop is due to a run-time error the termination state is -1.

3.1.2. The use of tags

A **tag** is a sequence of letters and digits, the first of which is a letter. All **tag**s defined by **rule-declaration**s, **pointer-initialization**s, **constant-description**s, **variable-description**s, **table-head**s (except those in **field-list-pack**s), **stack-head**s (except those in **field-list-pack**s), **file-description**s, **external-rule-description**s and **external-constant-description**s must differ from each other.

3.2. Rules

The declarations and applications of 'rules' constitute the mechanism for controlling the logical flow of the program. The **rule-declaration** defines *what* is to be done *if* the rule is called, whereas the application (in an **affix-form**) indicates *that* the rule is to be called.

A rule, when called, will either succeed or fail, according to criteria to be given in this manual and summarized in 3.9.2.

3.2.1. Rule-declarations

Each rule in the program must be declared exactly once, either in a **rule-declaration** or in an **external-rule-description** (for the latter see 5).

Syntax:

rule declaration:
typer, rule tag, [formal affix sequence], actual rule, point symbol.
typer:
action symbol; function symbol; predicate symbol; question symbol.
rule tag:
tag.

Example of a **rule-declaration**:

```
ACTION put string + ""file + table[] + >string - count:
    0 → count,
        (next - symb:
            string elem + table + string + count + symb,
                put char + file + symb, incr + count, :next;
            +
        ).
```

Here the **typer** is *ACTION*, the **rule-tag** is *put string*, the **formal-affix-sequence** is *+ ""file + table[] + >string* and the **actual-rule** is the rest, excluding the point but

including the – *count:*.

A **rule-declaration** defines the **actual-rule** to be of the type designated by **typer**, to be identified by the **rule-tag** and to have the formal affixes given by its **formal-affix-sequence**.

There are four types of rules: predicates, questions, actions and functions, each designated by the corresponding **typer** symbol. These four types arise from the fact that rules are differentiated on the basis of two mutually independent criteria:

a. a rule will *either* always succeed *or* be capable of failing, depending on the logical structure of the **actual-rule**,
b. a rule, when succeeding, may or may not have side-effects, again depending on the logical structure of the **actual-rule**.

These criteria are elaborated upon in 3.9.

A rule is a "predicate" if it can fail and has side-effects (the restrictions on the structure of rules prevent these side-effects from becoming effective if the rule fails).

A rule is a "question" if it can fail and has no side-effects.

A rule is an "action" if it will always succeed and has side-effects.

A rule is a "function" if it will always succeed and has no side-effects.

The type of a rule is checked against the logical construction of the **actual-rule**; if an action or function is found to be able to fail, an error message is given; in all other cases, if a discrepancy is found a warning is given.

Examples.

In each of the following examples the beginning of a **rule-declaration** is given, together with a summary of what the rule does. From this explanation it follows why the rule was declared with the given type.

PREDICATE digit + d>:	if the next character in the input file is a digit, it is delivered in *d*, the input file is advanced by one character (side-effect) and *digit* succeeds; otherwise it fails.
QUESTION is digit + >d:	if *d* is a digit the rule succeeds, otherwise it fails.
ACTION skip up to point:	the input file is advanced until the next character is a point.
FUNCTION plus + >x + >y + sum>:	the sum of *x* and *y* is delivered in *sum*.

3.2.2. Actual-rules

An **actual-rule** mentions the variables local to it and specifies one or more alternatives.

Syntax:

actual rule:
[local affix sequence], colon symbol, rule body.
rule body:
alternative series; classification.
alternative series:
alternative, [semicolon symbol, alternative series].
alternative:
last member; member, comma symbol, alternative.

Example of an **actual-rule**:

```
– d:
    digit + d, times + res + 10 + res,
        plus + res + d + res, integer 1 + res;
    +
```

Here the **local-affix-sequence** is – *d*, one **alternative** is

```
digit + d, times + res + 10 + res,
    plus + res + d + res, integer 1 + res
```

and + is another; *plus + res + d + res* is a **member** and + is a **last-member**.

When an **actual-rule** is executed (through a call (3.5) of the rule of which it is the **actual-rule**), the following takes place.

First space is made available on the run-time stack for the **local-affix**es, one location for each **local-affix** (see 3.3.3). Subsequently its **rule-body** is executed.

The execution of a **rule-body** implies the execution of its **alternative-series** or of its **classification**.

The execution of an **alternative-series** starts with a search to determine which of its **alternative**s applies in the present case. The applicable **alternative** is the (textually) first **alternative** whose 'key' succeeds. The "key" of an **alternative** is its first **member** or, if it has no **member**, its **terminator**. Thus, the key of the first **alternative** is executed: if it succeeds, the first **alternative** applies. Otherwise the key of the second **alternative** is executed: if it succeeds, the second **alternative** applies, etc. If none of the keys succeeds, the **alternative-series** fails.

The **alternative** found applicable is then elaborated further. Its key has already been executed. Now the rest of its **member**s and **last-member** are executed in textual order until one of two situations is reached:

either all its **member**s and its **last-member** have succeeded, in which case the **alternative-series** succeeds as well,

or a **member** or **last-member** fails: any (textually) following **member**s or **last-member** in this **alternative** will not be executed and the **alternative-series** fails.

If the **alternative-series** succeeded, the **actual-rule** succeeds; if it failed, the **actual-rule** fails.

For the execution of a **classification** see 3.8.

After the result of the **actual-rule** has thus been assessed, the space for the **local-affix**es is removed from the run-time stack.

Restrictions.

An **alternative-series** must satisfy the following restrictions:

a. If the key of an **alternative** cannot fail (3.9.2), the **alternative** must be the last one. This restriction ensures that all **alternatives** can, in principle, be reached. Violation of this restriction causes an error message.

b. If an **alternative** contains a **member** that has side-effects (see 3.9.1) this **member** may not, in the same **alternative**, be followed by a **member** that can fail (see 3.9.2).
This restriction ensures that the side-effects of a **member** cannot materialize if the **member** fails; this in turn ensures that the tests necessary to determine the applicable **alternative** in an **alternative-series** do not interfere with each other.
Violation of this restriction causes a warning. The user is urged either to reconsider the formulation of his problem or convince himself that the side-effects caused have no ill consequences.

3.2.3. Members

Members are the units of action in ALEPH. This action is a primitive operation, a call of a rule, or consists in its turn of other actions.

Syntax:

member:
    affix form; operation; compound member.
last member:
    member; terminator.

Example of a **member**:

```
(declaration sequence option − type − idf:
    declaration + type + idf, enter + type + idf,
        : declaration sequence option;
    +
)
```

This **member** is a **compound-member**, *declaration + type + idf* is an **affix-form**, *: declaration sequence option* is a **last-member**, as is +.

The notion **last-member** has been introduced in the syntax to ensure that a **terminator** will only occur last in an **alternative**.

3.3. Affixes

Formal and actual affixes constitute the communication between the caller of a rule and the rule called. Local affixes are a means for creating variables which are local to a given **rule-body**.

3.3.1. Formal-affixes

Syntax:

formal affix sequence:
    formal affix, [formal affix sequence].
formal affix:
    formal affix symbol, formal.
formal:
    formal variable; formal stack; formal table; formal file.

formal variable:
    [right symbol], variable tag, [right symbol].
formal table:
    [formal field list pack], table tag, sub bus.
formal stack:
    sub bus, [formal field list pack], stack tag, sub bus.
sub bus:
    sub symbol, bus symbol.
formal field list pack:
    open symbol, [field list], close symbol.
formal file:
    quote image, file tag.

Example of a **formal-affix-sequence**:

+ ""file + table[] + >string

The **formal-affix-sequence** defines the number and types of the **formal-affixes** of the rule it belongs to.

A **formal-variable** describes a variable. If the **formal-variable** starts with a **right-symbol** the variable has obtained a value from the calling rule; it has the attribute INITIALIZED. Otherwise it has the attribute UNINITIALIZED at the beginning of each alternative in the **actual-rule**.

If the **formal-variable** ends in a **right-symbol** its value will be passed back to the calling rule: it must have the attribute INITIALIZED at the end of each **alternative** of the **actual-rule** which does not end in a **jump**, **exit** or **failure-symbol**.

A **formal-stack** describes a stack. If the **formal-field-list-pack** is absent, the **formal-stack** is supposed to have one **selector**: the tag of this **selector** is the same as the tag of the **formal-stack** itself. For example, the **formal-affix** *[]list[]* has the same meaning as *[](list)list[]*.

A **formal-table** describes a table. If the **formal-field-list-pack** is absent, the **formal-table** is supposed to have one **selector**: the tag of this **selector** is the same as the tag of the **formal-table** itself.

A **formal-file** describes a file.

All **variable-**, **stack-**, **table-** and **file-tags** in a **formal-affix-sequence** must be different. They must also be different from the **rule-tag** that precedes the **formal-affix-sequence**.

3.3.2. Actual-affixes

Actual-affixes occur in **affix-forms** which cause the call of a rule. Each **actual-affix** corresponds to a **formal-affix** of that rule.

Syntax:

actual affix sequence:
    actual affix, [actual affix sequence].
actual affix:
    actual affix symbol, actual.
actual:
    source; list tag; file tag.

Example of an **actual-affix-sequence**:

*+ 511 + /?/ + alpha + beta*gamma[p] + <>list + ?*

In this example *511* is an **integral-denotation**, */?/* is a **character-denotation**, *alpha* may be a **file-tag**, *beta*gamma[p]* may be a **stack-element**, *<>list* is a **calibre** and *?* is a **dummy-symbol**.

Actual affixes derive their exact meanings from the corresponding **formal-affixes**. The interrelations are discussed in 3.5 (**affix-forms**) and in 3.4 (**transports**).

3.3.3. Local-affixes

Syntax:

local affix sequence:
    local affix, [local affix sequence].
local affix:
    local affix symbol, local variable.
local variable:
    variable tag.

Example of a **local-affix-sequence**:

– count

A **local-variable** describes a variable. Space for this **variable** is reserved on the run-time stack upon entry of the **actual-rule** or **compound-member** of which it is part. On exit from that **actual-rule** or **compound-member** this space is removed.

A **local-variable** has the attribute UNINITIALIZED at the beginning of each **alternative** of the **actual-rule** or **compound-member**. Its attribute must be INITIALIZED at the end of at least one **alternative**.

All **variable-tags** in a **local-affix-sequence** L must be different. Furthermore, all **variable-tags** in L must be different from:

a. all the **rule-tags**, if any, and all **variable-tags** in the **local-affix-sequences**, if any, of all the **compound-members**, if any, in which L is contained,
b. the **rule-tag** and all **variable-**, **stack-**, **table-** and **file-tags** in the **formal-affix-sequence**, if any, of the **rule-declaration** in which L occurs.

3.4. Operations

Syntax:

operation:
transport; identity; extension.

transport:
source, variable directive sequence.

source:
constant; variable.
constant:
plain value; table element.
plain value:
integral denotation; character denotation; constant tag; limit.
integral denotation:
[integral denotation], digit.
character denotation:
absolute symbol, character, absolute symbol.
variable:
variable tag; stack element; dummy symbol.
table element:
[selector, of symbol], table tag, sub symbol, source, bus symbol.
stack element:
[selector, of symbol], stack tag, sub symbol, source, bus symbol.

variable directive sequence:
variable directive, [variable directive sequence].
variable directive:
to token, variable.
to token:
minus symbol, right symbol.

identity:
source, equals symbol, source.

extension:
of symbol, field transport list, of symbol, stack tag.
field transport list:
field transport, [comma symbol, field transport list].
field transport:
source, selector directive sequence.
selector directive sequence:
selector directive, [selector directive sequence].
selector directive:
to token, selector.

Example of a **transport**:

*pnt → sel*list[q] → offset → ors*list[offset]*

Example of an **identity**:

*ect*list[pnt] = nil*

Example of an **extension**:

** pnt → sel, nil → ect → ors * list*

3.4.1. Transports

A **transport** can be considered a function, i.e., it has no (inherent) side-effects and will always succeed.

Its execution starts with the evaluation of its **source**. A **source** is evaluated as follows.

If the **source** is an **integral-denotation**, its value is the numerical value of the sequence of **digits**, considered as a number in decimal notation.

If the **source** is a **character-denotation**, its value is the numerical value of the **character** in the code used.

If the **source** is a **constant-tag** or a **variable-tag**, its value is the value of the constant or variable identified. If a formal or local variable is identified, it must have the attribute INITIALIZED.

If the **source** is a **stack-element** or a **table-element**, its value is determined as follows (see also 4.1.5 and 4.1.6).

The **source** between the **sub-symbol** and the **bus-symbol** is evaluated and its value is called P. We call the **stack-tag** or **table-tag** in front of the **sub-symbol** T, and the (global or formal) list identified by it L. We now consider the block in L that has an address equal to P (if no such block exists, there is an error); it is called B. Subsequently a selector S is determined: if the **of-symbol** is present, S is the **selector** in front of it; if the **of-symbol** is absent, S is T. (As an example, *list[p]* is equivalent to *list*list[p]*.) S must be a selector of L. Now, the value of the **stack-element** or **table-element** is the value in the block B identified by the selector S.

If the **source** is a **limit**, its value is described in 4.1.7.

If the **source** is a **dummy-symbol**, there is an error.

The value of the **source** is called V. Next the **variable-directive**s of the **transport** are executed in textual order. A **variable-directive** is executed as follows. ›

If its **variable** is a **variable-tag**, V is put in the location of the variable identified. If a formal or local variable is identified, this variable has the attribute INITIALIZED in the rest of the **alternative** in which the **transport** appears.

If its **variable** is a **stack-element**, the **source** between the **sub-symbol** and **bus-symbol** is evaluated and its value is called P. We call the stack identified by the **stack-tag** L. We now consider that block in L that has an address equal to P (if no such block exists, there is an error); it is called B. Subsequently a selector S is determined: if the **of-symbol** is present, S is the **selector** in front of it; if the **of-symbol** is absent, S is the **stack-tag**. S must be a selector of L. Now V is put in the location in the block B identified by the selector S.

If the **variable** is a **dummy-symbol**, the **variable-directive** is a dummy action.

Examples:

0 → cnt → res	now *cnt* and *res* are both zero
p → list[q] → q	the value of *p* is put in the location identified by *list*list[q]* and in (the location of) *q*
p → q → list[q]	the value of *p* is put in (the location of) *q* and then in the location identified by *list*list[q]* which is now the same as *list*list[p]*
list[p] → p → list[p]	the value of *list*list[p]* is put in *p* and then put in *list*list[p]* using the new value of *p*, with the result that now *list*list[p]* contains a pointer to itself

3.4.2. Identitys

An **identity** can be considered a question, i.e., it has no side-effects and may either succeed or fail.

Both its **source**s are evaluated as described above. If the two values are numerically equal the **identity** succeeds, otherwise it fails.

If the values represent numerical results the **identity** tests equality. If the values represent pointers to blocks in lists the **identity** tests whether the two blocks pointed at are the same, not whether they are equal (as this might imply complicated comparison criteria).

3.4.3. Extensions

An **extension** can be considered as an action, i.e., it has side-effects and will always succeed.

Call the stack identified by the **stack-tag** S. The **selector**s that appear in the **field-transport-list** must be selectors of S.

First the **source**s in the **field-transport**s are evaluated as described in 3.4.1 and their values remembered. Subsequently the stack S is extended to the right with one block B of empty locations (whence the name 'extension'); the number of locations in the block is equal to the calibre of S. Next the **field-transport**(s) are executed; a **field-transport** is executed by putting the value remembered for its **source** in the location(s) in B identified by its **selector**s.

No more than one value may be put in a given location in B; at the end of the **extension** all locations in B must have been given a value; if the stack is formal, the calibre of the actual stack must be equal to that of the formal stack.

Example: given a stack *st* declared as *[](sel, ect, ors)st*: then the **extension**

** 3 → ect, 5 → sel → ors * st*

will add the block *(5, 3, 5)* to *st* and *>>st* will be 3 higher than it was before.

3.5. Affix-forms

Syntax (see also 3.3.2):

affix form:
rule tag, [actual affix sequence].

Example:

string elem + tbl + str + cnt + symb

When an **affix-form** is executed, the rule identified by the **rule-tag** in the **affix-form** is called, as follows.

Relationships are set up between the **actual-affixes** as supplied by the **affix-form** and the **formal-affixes** as supplied by the **rule-declaration**. The correspondence between actual and formal affixes is decided from their order: the first actual corresponds to the first formal, the second actual to the second formal, and so on. The number of actuals must be equal to the number of formals.

The **actual** corresponding to a **formal-table** must be a **list-tag** identifying a (global or formal) stack or a (global or formal) table. All actions performed on the **formal** are executed directly on the **actual**. If the **formal** has a **field-list** the calibres of the **formal** and **actual** must be equal; the selectors may differ. If the **formal** has no **field-list**, no calibre match is required. Regardless of mismatches, the value delivered by the **calibre** ('*<>list*') is the calibre of the global list to which the **formal-table** corresponds, directly or indirectly.

The **actual** corresponding to a **formal-stack** must be a **stack-tag** identifying a (global or formal) stack. All actions performed on the **formal** are executed directly on the **actual**. If the **formal** has a **field-list** the calibres of the **formal** and **actual** must be equal; the selectors may differ. If the the **formal** has no **field-list**, no calibre match is required. Regardless of mismatches, the value delivered by the **calibre** is the calibre of the global stack to which the **formal-stack** corresponds, directly or indirectly.

The **actual** corresponding to a **formal-file** must be a **file-tag** identifying a (global or formal) file. All actions performed on the **formal** are executed directly on the **actual**.

First the copying part of the affix mechanism is put into operation: for each **formal** which is a **formal-variable** starting with a **right-symbol**, a **transport** is executed with the **actual** as its **source** and the **variable-tag** of the **formal** as its **variable**.

Subsequently, the **actual-rule** in the rule identified above is executed (see 3.2.2). If this **actual-rule** succeeds, the **affix-form** succeeds; if it fails, the **affix-form** fails.

If the **affix-form** succeeds the restoring part of the affix mechanism will be executed: for each **formal** that is a **formal-variable** ending in a **right-symbol**, a **transport** is executed with the **variable-tag** of the **formal** as its **source** and the **actual** as its **variable**, in the order in which the affixes appear.

Example:

Suppose the following rules are defined:

QUESTION if a: $ Some question $.
QUESTION if b: $ Another question $.
FUNCTION give value 1 + n>: 1 → n.
FUNCTION give value 2 + n>: 2 → n.
ACTION use value + >n: print + n.
ACTION print + >n: $ Some **actual-rule** *that prints the value of 'n' $.*

In the **actual-rule**

— *loc:*
if a, give value 1 + loc, use value + loc, print + loc;
if b, give value 2 + loc, use value + loc

loc has the attribute UNINITIALIZED at the colon and likewise at the first comma, INITIALIZED at the second comma because of the restoring done by the call of *give value 1*, and keeps the attribute INITIALIZED until the end of the **alternative**. Its value can be copied over to *use value* and *print*. At the beginning of the second alternative it still has the attribute UNINITIALIZED (*still* UNINITIALIZED, not *again* UNINITIALIZED, since, if the beginning of the second **alternative** is reached, the initialization in the previous **alternative** will not have taken place). It keeps the attribute UNINITIALIZED until the call of *give value 2* after (and by) which it obtains the attribute INITIALIZED. Its subsequent application in *use value* is correct.

The **actual-rule**

— *loc: if a, use value + loc, give value 1 + loc, print + loc*

is incorrect. *loc* still has the attribute UNINITIALIZED at the first comma and is then used as a **source** in the copying done by the call of *use value*.

3.6. Terminators

Syntax:

terminator:
jump; exit; success symbol; failure symbol.
jump:
repeat symbol, rule tag.
exit:
exit symbol, expression.

Examples of **terminators**:

: order
EXIT 16
+
—

Jumps.
The **rule-tag** after the **repeat-symbol** may be the **rule-tag** of the rule in which the **jump** occurs or the **rule-tag** of (one of) the **compound-member**(s) in which the **jump** occurs.

A **jump** to the **rule-tag** of a rule is an abbreviated notation of a call to that rule, with actual affixes that correspond to the original actual affixes. The abbreviation is only allowed if, after the execution of the call, no more members in the rule can be executed. This condition ensures that there will be no need for the 'recursive call' mechanism to be invoked.

Example:
The rule:

ACTION bad 1: a, (b; :bad 1), c; +.

is incorrect: after returning from *:bad 1* the **affix-form** *c* will be executed. If the , *c* is

removed, the rule is correct. Likewise the rule:

QUESTION bad 2: (a, b, :bad 2); c.

is incorrect: after unsuccessful returning from *:bad 2* the **affix-form** *c* will be executed. If the parentheses are removed, the rule is correct.

A **jump** to the **rule-tag** of a **compound-member** *C* causes this **compound-member** to be re-executed. The precise meaning can be assessed by decomposing (see 3.7) the rule until *C* turns into a rule. Then the above applies.

Exits.

The execution of an **exit** causes the entire program to be terminated. The termination state is equal to the value of the **expression** in the **exit**. An **exit** is a function.

Success- and **failure-symbols.**

The execution of a **success-symbol** always succeeds, the execution of a **failure-symbol** always fails. Neither has side-effects.

3.7. Compound-members

Compound-members serve to turn a (composite) **rule-body** into a single **member**.

Syntax:

compound member:
    open symbol, [local part, colon symbol], rule body, close symbol.
local part:
    rule tag, [local affix sequence]; local affix sequence.

Example:

```
(order - n:
    less + y + x, x → n, y → x, n → y;
    x = y, get next int + x, : order;
    +
)
```

A **compound-member** is an abbreviated notation for the call of a rule. Loosely speaking, the rule that is called has the same meaning as the **rule-body** of the **compound-member** and has all its non-globals as formal affixes. The call then calls that rule with these non-globals as actual affixes. The following statement expresses this more precisely.

A **rule-declaration** for the rule that is called can be derived from the **compound-member** *C* in the following way:

a. the **open-symbol** and **close-symbol** are removed,
b. a **point-symbol** is placed after the **rule-body**,
c. if the **local-part, colon-symbol** is absent, a **colon-symbol** is placed in front of the **rule-body**,
d. if the **rule-tag** is missing, a **rule-tag** is placed in front that produces a **tag** that is different from any other **tag** in the program,

e. a **formal-affix-sequence** is constructed (see below) and inserted after the **rule-tag**,
f. the 'type' of the **rule-body** is determined (see 3.9) and the corresponding **typer** (see 3.2.1) is placed in front of the **rule-tag**.

The **formal-affix-sequence** mentioned in e above is constructed as follows:

a. a list is made of all tags in the **rule-body** that do not refer to global items and do not occur in the **local-affix-sequence** of C, if present,
b. if the list is empty the **formal-affix-sequence** is empty,
c. for each tag in the list, if the corresponding item
   1. is used as a **source** (either directly or through the affix mechanism) and is used as a **variable** (either directly or through the affix mechanism), it is entered into the **formal-affix-sequence** preceded and followed by a **right-symbol**,
   2. is used as a **source** (either directly or through the affix mechanism), it is entered into the **formal-affix-sequence** preceded by a **right-symbol**,
   3. is used as a **variable** (either directly or through the affix mechanism), it is entered into the **formal-affix-sequence** followed by a **right-symbol**,
   4. is used as a **stack-tag** (or **table-tag**), it is entered into the **formal-affix-sequence** as a **formal-stack** (or **formal-table**) with the same **field-list-pack** as that of the corresponding (formal or actual) stack (or table),
   5. is used as a **actual-affix** where a file is required, it is entered into the **formal-affix-sequence** as a **formal-file**,
d. the items in the **formal-affix-sequence** are preceded by **formal-affix-symbol**s.

Example:
For the **compound-member**

```
(a[p] = 0, 0 → a[q]; plus + m + p + q)
```

where m is global, the **rule-declaration** runs:

```
ACTION zzgrzl + [](a)a[] + >p + >q>:
    a[p] = 0, 0 → a[q]; plus + m + p + q.
```

and the call is:

```
zzgrzl + a + p + q
```

This also implies that, if a **compound-member** fails, the changes it made to formal and local variables do not become effective. Compare

```
0 → n,
    (   (1 → n, −);
        n = 0, do something
    )
```

with

```
0 → n,
    (   spoil and fail + n;
        n = 0, do something
    )
```

where

QUESTION spoil and fail + n>: 1 → n, −.

Both cases behave in exactly the same way: the rule *do something* will be called.

The **rule-tag**, if any, of a **compound-member** *C* must be different from:

a. the **rule-tag**s, if any, and all the **variable-tag**s in the **local-affix-sequence**s, if any, of all the **compound-member**s, if any, in which *C* occurs,
b. the **rule-tag** and all the **variable-tag**s, **stack-tag**s, **table-tag**s and **file-tag**s in the **formal-affix-sequence**, if any, of the **rule-declaration** in which *C* occurs.

3.8. Classifications

A **classification** is similar to an **alternative-series** in that both specify a series of **alternative**s, only one of which will eventually apply. The difference is twofold: in a **classification** exactly one **alternative** applies (as opposed to one or zero in an **alternative-series**), and the choice of the pertinent **alternative** is based on a single run-time value (as opposed to the successive execution of keys). **Classification**s allow fast selection of **alternative**s at the cost of a less versatile selection mechanism.

Syntax:

classification:
    classifier box, class chain.
classifier box:
    box symbol, classifier, box symbol.
classifier:
    source.
class chain:
    class, semicolon symbol, class chain; last class.
class:
    area, comma symbol, alternative.

area:
    sub symbol, zone series, bus symbol.
zone series:
    zone, [semicolon symbol, zone series].
zone:
    [expression], up to symbol, [expression]; expression; list tag.
last class:
    class; alternative.

Example 1:

```
(n: get + char,
        (    = char =
             [/0/ : /9/], dgt → type;
             [/a/ : /z/; /a/ + cap : /z/ + cap], ltr → type;
             [/+/; /−/; /×/; ///], op → type;
             [0; 127], : n;
                  err → type
        )
)
```

Example 2:

```
= tag =
[var decl], handle variable + tag;
[macro decl], handle macro call + tag;
[rout decl], handle routine call + tag;
    handle bad tag + tag
```

The execution of a **classification** starts with the evaluation of the **source** in its **classifier-box**. The resulting value is called V. Now the **areas** in the **classification** are searched in textual order for an **area** in which V belongs. If such an **area** is found, the **alternative** following it applies and is executed (see 3.2.2). If there is no such **area**, the **last-class** must be an **alternative**, which then applies and is executed. Otherwise there is an error.

V belongs in a given **area** if it belongs in any of its constituent **zones**. Whether V belongs in a given **zone** is determined as follows.

If the **zone** is an **expression** E then V belongs in that **zone** if it is equal to the value of E.

If the **zone** contains an **up-to-symbol** it is designated by two boundaries. The left boundary L is the value of the **expression** in front of the **up-to-symbol** or, if it is missing, the value of *min int*. The right boundary R is the value of the **expression** after the **up-to-symbol** or, if it is missing, the value of *max int*. V belongs to the given **zone** if $L \leqslant V \leqslant R$.

If the **zone** is a **list-tag**, this **list-tag** must identify a global (*not* formal) list. V belongs in the **zone** if it is an address in the virtual address space (4.1.4) of the list.

Areas may coincide partially or totally; the textually first **area** takes precedence.

The exact size and location of all **zones** is known at compile time; this information can be utilized by the compiler.

A **classification** can fail if at least one of its **alternatives** can fail, it has side-effects if at least one of its **alternatives** has side-effects.

3.9. Criteria for side-effects and failing

When a list of conditions is given in this paragraph, the requirements for this list are fulfilled if at least one of the conditions is fulfilled.

3.9.1. Criteria for side-effects

In essence a rule "has side-effects" if it changes global information.

A rule has side-effects if its **rule-body** has side-effects.

A **rule-body** (i.e., an **alternative-series** or a **classification**) has side-effects if it contains at least one **member** that has side-effects.

A **member** has side-effects if

1. it is an **affix-form** that has side-effects,
2. it is a **transport** that has side-effects,

3. it is an **extension** or
4. it is a **compound-member** the **rule-body** of which has side-effects.

An **affix-form** has side-effects if
1. the rule called is an action or a predicate or
2. the restoring part of the affix mechanism (see 3.5) causes a **transport** that has side-effects.

A **transport** has side-effects if (one of) its **variable**(s) identifies a global variable or is a **stack-element**.

3.9.2. Criteria for failure

A **member** can fail if
1. it is an **affix-form** the rule of which is a predicate or question,
2. it is an **identity** or
3. it is a **compound-member** the **rule-body** of which can fail.

A **terminator** can fail if
1. it is a **failure-symbol** (−) or
2. it is a **jump** to a rule or **compound-member** that can fail.

A **rule-body** can fail if its **alternative-series** or **classification** can fail.

An **alternative-series** can fail if
1. the key of its last **alternative** can fail or
2. it contains an **alternative** that contains a **member** or **terminator**, other than its key, that can fail.

A **classification** can fail if it contains a **member** that can fail.

4. DATA

The basic way of representing information in ALEPH is through integers. There are four integer-based data types:
- integers ('constants'),
- locations that contain integers ('variables'),
- ordered lists of integers ('tables'), and
- ordered lists of locations that contain integers ('stacks').

Integers used in data declarations can be given in the form of expressions.

The basic way of routing information into and out of the program is through files. There are two types of files:
- 'charfiles', files containing only integers that correspond to characters, and
- 'datafiles', files containing pointers to prescribed stacks and tables and/or integers in a prescribed range.

There are three primitive actions on integer-based data: **transport**s, **identity**s and **extension**s. Additional integer handling can be done through externals.

There are no primitive actions on files: all file handling is done through externals.

Syntax of **data-declaration**:

data declaration:
constant declaration;
variable declaration;
stack declaration;
table declaration;
file declaration.

4.1. Integer-based data

Since all integer-based data can be initialized through expressions, these will be treated first.

4.1.1. Expressions

Syntax:

expression:
[plus minus], term; expression, plus minus, term.
term:
[term, times by], base.
base:
plain value; expression pack.
expression pack:
open symbol, expression, close symbol.
plus minus:
plus symbol; minus symbol.
times by:
times symbol; by symbol.

Examples:

*−3 + 5 * byte size*
line width/2
*((/e/ + 1) * char size + /n/ + 1) * char size + /d/ + 1*

The value of an **expression** is the integral value that results from evaluating the **expression** according to the standard rules of algebra.

The result of an integer division $n = p/q$ ($q \neq 0$) is a value n such that $p - n \times q$ is non-negative and minimal (so, e.g., $7/3=2$, $7/(-3)=-2$, $(-7)/3=-3$ and $(-7)/(-3)=3$).

A **constant-tag** defined in a user-defined **external-constant-declaration** cannot be used in an **expression**.

The **list-tag** in a **min-limit** or **max-limit** (see 4.1.7) used in an **expression** must identify a (global) table, i.e., limits of stacks cannot be used in **expressions**.

4.1.2. Constants

A "constant" consists of a **constant-tag** and an integral value. The relation between tag and value is set up through a **constant-declaration** and cannot be changed afterwards.

Syntax:

The **list-tag** in a **min-limit** or **max-limit** (see 4.1.7) used in an **expression** must identify a (global) table, i.e., limits of stacks cannot be used in **expressions**.

4.1.2. Constants

A "constant" consists of a **constant-tag** and an integral value. The relation between tag and value is set up through a **constant-declaration** and cannot be changed afterwards.

Syntax:

constant declaration:
    constant symbol, constant description list, point symbol.
constant description list:
    constant description, [comma symbol, constant description list].
constant description:
    constant tag, equals symbol, expression.
constant tag:
    tag.

Example:

CONSTANT mid page = line width/2, line width = 144.

The value of the **expression** must not depend on the **constant-tag** being declared. That is,

CONSTANT p = q, q = 2 − p.

is not allowed.

Constants can be used in **expressions** and in **sources**.

4.1.3. Variables

A "variable" consists of a **variable-tag** and a location; the location may or may not contain a value. If it contains a value the variable "has" that value. The contents of a location may be changed. Once a location has obtained a value it can never become empty again.

A global variable is declared in a **variable-declaration**.

A formal variable originates from a **formal-affix-sequence**.

A local variable originates from a **local-affix-sequence**.

Syntax of **variable-declaration**:

variable declaration:
    variable symbol, variable description list, point symbol.
variable description list:
    variable description, [comma symbol, variable description list].
variable description:
    variable tag, equals symbol, expression.
variable tag:
    tag.

Examples:

4.1.4. The address space

In addition to constants and variables, lists of constants ('tables') and lists of variables ('stacks') exist. Stacks and tables together are called "lists". The items in these lists are identified by unique addresses which are represented by integral values. These values range from a (large) negative number to a (large) positive number: this range is called the "address space".

The lists are described as running from left to right.

Example:

On a 16-bit machine the address space could be thought of as a list of 2^{16} (65536) locations, the addresses of which run from -2^{15} (-32768) at the left to $2^{15}-1$ (32767) at the right. The question whether all these locations actually exist in memory is at this point immaterial: it is only the addressability of a location that is secured here.

For a given program the address space is divided into chunks, one for each list. Consequently, an address uniquely identifies not only a location but also the list it belongs to. A chunk of address space belonging to a list is called its "virtual address space". Generally only a part of the virtual address space is in use: this part is called the "actual address space". From the language specifications it follows that an actual address space is always a contiguous list of locations or values.

The user has no direct control over the way in which the address space is divided and addresses are assigned. This is done as follows:

a. Deleted; see 5.2.4 for *nil* and *nil table*.
b. For each table or stack without **size-estimate** L the size of its actual address space is calculated from its **filling-list** and L is given a virtual address space of exactly the same size.
c. For each stack with an **absolute-size** a virtual address space of that size is reserved.
d. The remainder of the virtual address space is distributed over the rest of the stacks, proportionally to their **relative-size**s.

For each list L the right-most address in its virtual address space is called "virtual max limit", the left-most address in its virtual address space minus one plus the 'calibre' of L is called "virtual min limit"; the size of its actual address space is calculated from its **filling-list** and the actual address space is positioned at the left end in the virtual address space. The 'max limit' of L is made equal to the right-most address in the actual address space; the 'min limit' of L is made equal to the 'virtual min limit'.

If the actual address space has length zero, the 'max limit' of L is equal to the 'min limit' minus the 'calibre' of L.

The virtual and actual address space of a table are fixed (and equal) for the duration of the program.

Example:

Suppose a virtual address space of 5 bits, i.e. the addresses range from -16 to 15. If the following declarations (see 4.1.5 and 4.1.6) occur in the program:

```
TABLE powers = (1, 10, 100, 1000).
STACK [= 5 =] digits = (0),
    [ 30 ] stack,
    [ 50 ] (num, denom) rationals = ((355, 113): pi, (191, 71): e).
```

the virtual address space could have the following layout:

address:	contents:	belongs to:	selector:	pointer:
−16	— —	— —	— —	*nil*
−15	1	*powers*	*powers*	*<<powers*
−14	10	"	"	
−13	100	"	"	
−12	1000	"	"	*>>powers*
−11	0	*digits*	*digits*	*<<digits, >>digits*
−10	— —	"	"	
−9	— —	"	"	
−8	— —	"	"	
−7	— —	"	"	*>>stack*
−6	— —	*stack*	*stack*	*<<stack*
−5	— —	"	"	
−4	— —	"	"	
−3	— —	"	"	
−2	— —	"	"	
−1	— —	"	"	
0	— —	"	"	
1	— —	"	"	
2	355	*rationals*	*num*	
3	113	"	*denom*	*<<rationals, pi*
4	191	"	*num*	
5	71	"	*denom*	*>>rationals, e*
6	— —	"	*num*	
7	— —	"	*denom*	
8	— —	"	*num*	
9	— —	"	*denom*	
10	— —	"	*num*	
11	— —	"	*denom*	
12	— —	"	*num*	
13	— —	"	*denom*	
14	— —	"	*num*	
15	— —	"	*denom*	

(For the notation used see 4.1.5 through 4.1.7).

ALEPH allows the user to extend a stack towards the right (raising the 'max limit') through an extension (3.4.3); to remove items from the right of a stack through a call of *unstack*, *unstack n*, *scratch* or *delete* (5.2.4) after which the discarded address space can be reclaimed (but not the values in it) through an extension; and to remove items from the left of a stack through a call of *unqueue* or *unqueue n* (5.2.4) after which the discarded address space is irrevocably lost.

Through the use of these features a stack can be operated in stack fashion ('add to right end'/'remove from right end') or in queue fashion ('add to right end'/'remove from left end'). Queue-operation consumes virtual address space but in most implementations virtual address space will be virtually unlimited.

Usually an actual address space corresponds to a physical space that is in the physical memory of the computer used. The physical space is completely invisible to the user except perhaps in efficiency considerations. Parts of it may be in main memory, managed by some re-allotment scheme, parts of it may be on background memory.

4.1.5. Tables

Tables originate from **table-declarations**.

Syntax:

table declaration:
    table symbol, table description list, point symbol.
table description list:
    table description, [comma symbol, table description list].
table description:
    table head, equals symbol, filling list pack.
table head:
    [field list pack], table tag.
table tag:
    tag.

field list pack:
    open symbol, field list, close symbol.
field list:
    field, [comma symbol, field list].
field:
    selector chain.
selector chain:
    selector, [equals symbol, selector chain].
selector:
    tag.

filling list pack:
    open symbol, filling list, close symbol.
filling list:
    filling, [comma symbol, filling list].
filling:
    single block; compound block; string filling.

single block:
    expression, [pointer initialization].

compound block:
expression list proper pack, [pointer initialization].
pointer initialization:
colon symbol, constant tag.
expression list proper pack:
open symbol, expression list proper, close symbol.
expression list proper:
expression, comma symbol, expression list.
expression list:
expression, [comma symbol, expression list].

string filling:
string denotation, [pointer initialization].
string denotation:
quote symbol, [string item sequence], quote symbol.
string item sequence:
string item, [string item sequence].
string item:
non quote item; quote image.
quote image:
quote symbol, quote symbol.

Examples:

```
TABLE messages =
    (   "tag undefined": bad tag,
        "wrong number of parameters": wrong parameter,
        "quote "" where not allowed": bad quote
    ).

TABLE hexadec =
    (   /0/, /1/, /2/, /3/, /4/, /5/, /6/, /7/,
        /8/, /9/, /a/, /b/, /c/, /d/, /e/, /f/
    ).

TABLE (wind, next) four winds =
    (   (north wind, east): north,
        (east wind, south): east,
        (south wind, west): south,
        (west wind, north): west
    ).
```

4.1.5.1. The table-head

A "table" is a sequential list of integral values. For referencing purposes these values are numbered sequentially. The numbers which can be used as addresses are chosen by the compiler and are unique to the given table, i.e., no two integral values in tables have the same address. The right-most item in the table has the largest address, which is known as the 'max limit' of the table. The left-most item has the smallest address, the smallest address minus one plus the calibre is known as the 'min

limit' of the table. Consequently the number of values in the table is 'max limit' − 'min limit' + 'calibre'.

If the **field-list-pack** is missing, a **field-list-pack** of the form:

open symbol, table tag, close symbol

where the **table-tag** is the same as that of the **table-head**, is supposed to be present. For example:

TABLE messages = ...

means

TABLE (messages) messages = ...

4.1.5.2. The field-list-pack and the filling-list

The following applies to tables and stacks alike.

All **tag**s in a **field-list-pack** must differ one from another.

The "calibre" C of a list is the number of **field**s in the **field-list-pack**. The list is considered to be subdivided into blocks of length C; this implies that 'max limit' − 'min limit' is an integral multiple of C. The address of the right-most item in a block is considered the address of that block. Each value in a block can be referenced through a **selector**: the **field**s in the **field-list-pack** correspond, in that order, to the values in the block. A **field** is identified by one of its **selector**s.

The values in the list are specified in the **filling-list-pack**. Each **filling** in the **filling-list-pack** corresponds to one or more blocks in the list: the first block produced by the **filling-list-pack** corresponds to the left-most block in the list, and so on.

If the **filling** is a **single-block**, the calibre of the list must be 1. It gives rise to one block; the value in the block is the value of the **expression**. If a **pointer-initialization** is present the **constant-tag** in it is defined as having the value of the address of the block.

If the **filling** is a **compound-block**, the number of **expression**s in it must be equal to the calibre of the list. The values in the block are the values of the **expression**s. If a **pointer-initialization** is present the **constant-tag** in it is defined as having the value of the address of the block.

If the **filling** is a **string-denotation**, the calibre of the list must be 1. It gives rise to one or more blocks of one value each that describe the given string in a machine-dependent way. If a **pointer-initialization** is present the **constant-tag** in it is defined as having the value of the largest address in the generated list of blocks.

The string denoted by a **string-denotation** consists of the characters which are the representations of its **string-item**s, if any, except that for each **quote-image** the representation of the **quote-symbol** is taken. Spaces are considered **string-item**s, new-line control characters are not, since the dividing into lines is done through the charfile-handling externals (see 5.2.5).

Example 1:

The **table-declaration** for *four winds* (example 3 above) gives rise to the following list:

address:	selector:	value:

	wind	*north wind*
north	*next*	*east*
	wind	*east wind*
east	*next*	*south*
	wind	*south wind*
south	*next*	*west*
	wind	*west wind*
west	*next*	*north*

and *wind * four winds [next * four winds [west]]* has the value *north wind.*

Example 2:
The **table-declaration**

TABLE strings = ("abcdefg": letters, "01234": digits)

could in some version on some computer generate:

address:	selector:	value:
	strings	13 14 15 16
	"	17 20 21 00
letters	"	00 07 00 02
	"	01 02 03 04
	"	05 00 00 00
digits	"	00 05 00 02

A **table-tag** can be used in a **table-element** or a **limit**, or as an **actual** in an **affix-form**, or to indicate a **zone** in a **classification** or **file-description**.

4.1.6. Stacks

Stacks originate from **stack-declarations**.

Syntax:

stack declaration:
stack symbol, stack description list, point symbol.
stack description list:
stack description, [comma symbol, stack description list].
stack description:
stack head, [equals symbol, filling list pack].

stack head:
[size estimate], [field list pack], stack tag.
size estimate:
relative size; absolute size.
relative size:
sub symbol, expression, bus symbol.
absolute size:
sub symbol, box symbol, expression, box symbol, bus symbol.
stack tag:
tag.

Examples:

STACK [= line width =] (char) print line.

STACK [40] (tag pnt, left, right) idf list =
$ the following **filling-list-pack** *describes a binary tree*
$ containing the standard identifiers of ALGOL 60.
((exp st, cos, sign): exp,
(abs st, nil, arctan): abs,
(arctan st, nil, nil): arctan,
(cos st, abs, entier): cos,
(entier st, nil, nil): entier,
(ln st, nil, nil): ln,
(sign st, ln, sin): sign,
(sin st, nil, sqrt): sin,
(sqrt st, nil, nil): sqrt
).

A "stack" is a (possibly empty) sequential list of locations that contain integral values. The structure of this list and its addressing scheme is parallel to that of a table. The initial values in the locations are determined by the **filling-list-pack** in a way analogous to that used for tables. The 'max limit' is equal to the address of the right-most location, the 'min limit' is equal to the address of the left-most location minus one plus the 'calibre' of the stack. Again these values are chosen by the compiler and are unique to the given stack.

The values of the **expression**s in the **size-estimate**s must not depend, directly or indirectly, on the value of any **contstant-tag** defined in a **pointer-initialization**.

The values in the locations in a stack can be altered by transporting (3.4) a value into an element of that stack. For ways of changing the size of a stack, see 4.1.4.

A **stack-tag** can be used in a **stack-element**, a **limit** or an **extension**, or as an **actual** in an **affix-form**, or to designate a **zone** in a **classification** or **file-description**.

4.1.7. Limits

Syntax:

limit:
min limit; max limit; calibre.
min limit:
min token, list tag.
max limit:
max token, list tag.
calibre:
calibre token, list tag.

list tag:
stack tag; table tag.
min token:
left symbol, left symbol.
max token:
right symbol, right symbol.
calibre token:
left symbol, right symbol.

Examples:

<<stack, >>table, <>blocked

A **min-limit** (**max-limit**, **calibre**) has the value of the 'min limit' ('max limit', 'calibre') of the list identified by the **list-tag**.

The value of a **limit** is a constant in that it cannot be changed by a **transport**. However, the values of the **min-limit** and the **max-limit** of a stack may change as a consequence of actions which change the size of that stack. The values of the **min-limit** and the **max-limit** of tables and of the **calibre**s of all lists are invariable.

4.2. Files

Files originate from **file-declaration**s. They can be prefilled by the operating system (input files) or postprocessed by the operating system (output files) or both (I/O files) or neither (scratch files).

Syntax:

file declaration:
file typer, file description list, point symbol.
file typer:
charfile symbol; datafile symbol.
file description list:
file description, [comma symbol, file description list].
file description:
file tag, [area],
equals symbol, [right symbol], string denotation, [right symbol].
file tag:
tag.

Examples:

CHARFILE printer = "output">, backward lines = >"qelet,invert".

DATAFILE tagfile[tag; link; 0:] = >"systags">,
bin[0:4095] = "12row,bin">, overflow[:] = "☜☞qxz".

A **file-description** declares a "file" of the type designated by the **file-typer**. If the first **right-symbol** is present, the file is prefilled by the operating system (but it may still be empty); if the second **right-symbol** is present, the file will be postprocessed by the operating system (but it may be empty).

The (implementation-dependent) **string-denotation** must contain enough information to enable the operating system to manipulate the file in the desired way. It might

for example contain: the external file name, allocation information, the names of routines to do the prefilling and postprocessing, etc.

ALEPH contains no explicit file handling statements: all file handling is done through (standard) externals (see 5.2.5). When a file is used for writing, each item offered must belong in the **area** given in the **file-description**; when a file is used for reading, each item delivered will belong in the given **area**. If no **area** is supplied, the **area** *[:]* is assumed.

Files are read and written sequentially. They can be reset to the beginning of the file and be reread or rewritten. The file ends after the last item written or else after the last item produced by the preprocessing.

4.2.1. Charfiles

A "charfile" is a list of "lines". A 'line' consists of a control integer and a (possibly empty) sequence of characters. Characters are values in the **area** *[0:max char]*, control integers are values outside that **area**. Four control integers are predefined in the compiler (see 5.2.5): *new line*, *same line*, *rest line* and *new page*. These control integers can be used by the pre- and post-processing to reconcile the system requirements with the ALEPH requirements. If the file is eventually postprocessed towards a printer, lines of the type *new line* will be printed on new lines, those of the type *same line* will be printed over the previous line and those of type *new page* will be printed on the first line of a new page; *rest line* serves administration purposes only. Analogous effects should be defined for other devices, as far as the analogy will stretch.

Example:
A file containing

a&b=b&a

would consist of two lines:

new line, /a/, /&/, /b/, /=/, /b/, /&/, /a/
same line, / /, / /, / /, /_/.

The standard externals allow two ways of processing a charfile.

a. linewise: each call of *PREDICATE get line + ""charfile + []stack[] + cint>* puts the next line on *stack* (the last character on the line is the right-most item in the stack) and yields the control integer in *cint*. It will fail if there is no next line.
b. characterwise: each call of *PREDICATE get char + ""charfile + char>* yields the next item from the *charfile* (control integers and characters alike). It will fail if there is no next item.

The **area** in the **file-description** of a charfile pertains to the values of the characters only. If present, the **area** must only specify values that belong in *[0:max char]*, e.g. *[0:1]*.

4.2.2. Datafiles

A "datafile" is a list of "data-items". A data-item consists 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 is the name of a list, in which case the integer value is an offset from the left end of that list.

A data-item is written on a datafile by a call of *ACTION put data + ""file + >item + >type*. The data-item is constructed from the *item*- and *type*-parameters and from the **area** in the **file-description** of the *file* in the following way.

If the *type* is *numerical*, there must be a **zone** in the **area** which is not a tag identifying a list, such that the value of *item* belongs in that **zone**. The data-item then consists of the value of *item* and the indication NUMERICAL.

If the *type* is *pointer*, the value of *item* must be an address in the virtual address space of a list whose **list-tag** is a **zone** in the **area**. The data-item then consists of the offset from the left end of that list and the name of the list.

A data-item is read from a datafile by a call of *PREDICATE get data + ""file + item> + type>*. If there is still a data-item on *file*, it is read and the *item* and *type* are reconstructed from it (see above). If there are no more data-items on the datafile, the predicate fails.

Datafiles can be used to transfer information from one ALEPH-program to another. Pointers to lists which are in different positions in both programs are adjusted automatically during the transfer.

Note: in practice it is not necessary to record the list name with every item. It is enough to have one bit per item and one translation table for the whole file.

Example:
Suppose the **file-declaration**:

```
DATAFILE tag file[tag; list; 0: ] = >"systags">.
```

Then *put data* for this file can be visualized as:

```
ACTION put data + ""file + >item + >type:
$ For 'file' = 'tag file' only:
    type = pointer,
        (    = item =
             [tag], minus + item + <<tag + item,
                  write data item + item + tag name;
             [list], minus + item + <<list + item,
                  write data item + item + list name;
             error + bad item
        );
    type = numerical,
        (    = item =
             [0: ], write data item + item + NUMERICAL;
             error + bad item
        );
    error + bad type.
```

Here the (imaginary) *write data item + >val + >ind* would write a data-item consisting of *val* and *ind* on the file *tag file*.

5. EXTERNALS

External rules, tables and constants can be used in the same way as internally declared rules, tables and constants. An external rule differs from an 'internal' rule in that its body is not given in the program but is instead obtained from external sources. In the same way the values of external tables and constants are obtained from external sources. The necessary information can be supplied by the user through external means ('user' externals, section 5.1) in which case the name of the item and some of its properties must be declared in the program, or it is supplied automatically by the compiler ('standard' externals, section 5.2) in which case there is no explicit declaration at all.

5.1. User externals

Syntax:

external declaration:
external rule declaration;
external table declaration;
external constant declaration.

external rule declaration:
external symbol, typer, external rule description list, point symbol.
external rule description list:
external rule description,
[comma symbol, external rule description list].
external rule description:
rule tag, [formal affix sequence], equals symbol, string denotation.

external table declaration:
external symbol, table symbol,
external table description list, point symbol.
external table description list:
external table description,
[comma symbol, external table description list].
external table description:
table head, equals symbol, string denotation.

external constant declaration:
external symbol, constant symbol,
external constant description list, point symbol.
external constant description list:
external constant description,
[comma symbol, external constant description list].
external constant description:
constant tag, equals symbol, string denotation.

Example:

EXTERNAL FUNCTION convert to hash + t[] + >p + h> =
"subr, convertt".
EXTERNAL TABLE conv 2 ebcdic = "addr, conv2ebc".
EXTERNAL CONSTANT max ebcdic = "cons, maxebcdi".

An **external-rule-description** defines a rule to be of the type given by the preceding **typer**, to be known internally under the name given by the **rule-tag** and externally by the **string-denotation**, and to have affixes as shown by the **formal-affix-sequence**. A call to such a rule will result in implementation-dependent actions; it is the implementer's responsibility to see to it that these actions are in accordance with the type of the rule and that no side-effects will occur when a call of the rule fails.

An **external-table-description** defines a table to be known internally under the name given by the **table-tag** in the **table-head** and externally by the **string-denotation**, and to have the selectors given by the **field-list-pack**. An application of this table will result in implementation-dependent actions.

An **external-constant-description** defines a constant to be known internally under the name given by the **constant-tag** and externally by the **string-denotation**. An application of this constant will result in implementation-dependent actions.

5.2. Standard externals

Standard externals can be used in all programs without further notice. Their names can be redeclared by the user.

5.2.1. Integers

For those data considered to be integers, the following standard externals are available.

- *CONSTANT zero, one, max int, min int, int size.*
  zero has the value 0, *one* has the value 1. *max int* has the value of the largest integer in the given implementation, and *min int* has the value of the smallest (most negative) integer in the given implementation. *int size* is the number of decimal digits necessary to represent *max int*.
- *FUNCTION add + >a + >b + head> + tail>.*
  The double-length sum of *a* and *b* is given in *head* and *tail*: $a + b = head \times (max\ int + 1) + tail$, such that $|head|$ is minimal.
- *FUNCTION subtr + >a + >b + head> + tail>.*
  The double-length difference of *a* and *b* is given in *head* and *tail*: $a - b = head \times (max\ int + 1) + tail$, such that $|head|$ is minimal.
- *FUNCTION mult + >a + >b + head> + tail>.*
  The double-length product of *a* and *b* is given in *head* and *tail*: $a \times b = head \times (max\ int + 1) + tail$, such that $|head|$ is minimal.
- *FUNCTION divrem + >a + >b + quot> + rem>.*
  The quotient and remainder of the integer division of *a* by *b* is given in *quot* and *rem*: $a = b \times quot + rem$, such that *rem* is non-negative and minimal. *b* must not be zero.

- *FUNCTION plus + >a + >b + c>.*
  The sum of *a* and *b* is given in *c*.
- *FUNCTION minus + >a + >b + c>.*
  The difference of *a* and *b* (i.e., $a - b$) is given in *c*.
- *FUNCTION times + >a + >b + c>.*
  The product of *a* and *b* is given in *c*.
- *FUNCTION incr + >x>.*
  The value of *x* is increased by 1.
- *FUNCTION decr + >x>.*
  The value of *x* is decreased by 1.

- *QUESTION less + >p + >q.*
  Succeeds if *p* is less than *q*, fails otherwise.
- *QUESTION lseq + >p + >q.*
  Succeeds if *p* is less than or equal to *q*, fails otherwise.
- *QUESTION more + >p + >q.*
  Succeeds if *p* is more than *q*, fails otherwise.
- *QUESTION mreq + >p + >q.*
  Succeeds if *p* is more than or equal to *q*, fails otherwise.
- *QUESTION equal + >p + >q.*
  Succeeds if *p* is equal to *q*, fails otherwise. It is identical to '$p = q$'.
- *QUESTION noteq + >p + >q.*
  Succeeds if *p* is not equal to *q*, fails otherwise.

- *ACTION random + >p + >q + r>.*
  A pseudo-random number between *p* and *q* is given in *r*: $p \leqslant r \leqslant q$. The value of *r* is derived from an element in a uniformly distributed sequence of random numbers. The next call of *random* will derive its output value from the next number in that sequence, etc.
- *ACTION set random + >n.*
  n determines in some way the position in the sequence of random numbers mentioned above, from which the next call of *random* will obtain its output value.
- *ACTION set real random.*
  The position in the sequence of random numbers used by *random* is determined in an unpredictable way.

- *QUESTION sqrt + >a + root> + rem>.*
  If *a* is non-negative, *sqrt* succeeds; the square root and remainder of *a* are yielded such that $a = root \times root + rem$, and *rem* is non-negative and minimal. Otherwise it fails.

- *FUNCTION pack int + from[] + >n + int>.*
  The right-most *n* elements in the list *from* must be integer values corresponding to characters that correspond to digits. The digits thus indicated are considered as the decimal notation of an integer, and the value of this integer is yielded in *int*. A check on integer overflow is performed.
  Example: if the 4 right-most elements of *st* are:

  /0/, /2/, /7/, /3/

  then a call of *pack int + st + 4 + res* will assign the value 273 to *res*.
- *ACTION unpack int + >int + []st[].*
  The absolute value of *int* is written in decimal notation in *int size* digits, and *st* is extended with the integer values of the digits thus obtained, in left-to-right order.

The following externals are recommended.

- *FUNCTION date + year> + month> + day>.*
  The year, month and day are yielded in *year*, *month* and *day*.
- *FUNCTION time + amount>.*
  If two calls of *time* yield *amount 1* and *amount 2* respectively, then *amount 2 − amount 1* is in some way indicative for the time spent by the program between these two calls.

5.2.2. Words

For those data that are considered to be arrays of bits (words), the following standard externals are available.

- *CONSTANT word size.*
  The bits in a word are numbered (from left to right) from *word size* − 1 to 0.
- *CONSTANT false, true.*
  The value of *false* is 0, that of *true* is 1.
- *FUNCTION bool invert + >a + b>.*
  A word is yielded in *b* that contains a 1 in those positions where *a* contains a 0, and a 0 otherwise.
- *FUNCTION bool and + >a + >b + c>.*
  A word is yielded in *c* that contains a 1 in those positions where both *a* and *b* contain a 1, and a 0 otherwise.
- *FUNCTION bool or + >a + >b + c>.*
  A word is yielded in *c* that contains a 1 in those positions where either *a* or *b* or both contain a 1, and a 0 otherwise.
- *FUNCTION bool xor + >a + >b + c>.*
  A word is yielded in *c* that contains a 1 in those positions where *a* and *b* differ, and a 0 otherwise.

- *FUNCTION left circ + >x> + >n.*
  The bit-array in *x* is shifted *n* positions to the left; bits leaving the word on the left are re-introduced on the right. It is required that $0 \leqslant n \leqslant$ *word size*.

- *FUNCTION left clear + >x> + >n.*
  The bit-array in x is shifted n positions to the left; bits leaving the word on the left are discarded and 0s are introduced on the right. It is required that $0 \leqslant n \leqslant$ *word size.*
- *FUNCTION right circ + >x> + >n.*
  The bit-array in x is shifted n positions to the right; bits leaving the word on the right are re-introduced on the left. It is required that $0 \leqslant n \leqslant$ *word size.*
- *FUNCTION right clear + >x> + >n.*
  The bit-array in x is shifted n positions to the right; bits leaving the word on the right are discarded and 0s are introduced on the left. It is required that $0 \leqslant n \leqslant$ *word size.*
- *QUESTION is elem + >x + >n.*
  Succeeds if the n-th bit in x is a 1, fails otherwise. It is required that $0 \leqslant n <$ *word size.*
- *QUESTION is true + >x.*
  Succeeds if x contains at least one 1, fails otherwise.
- *QUESTION is false + >x.*
  Succeeds if x contains only 0s, fails otherwise.
- *FUNCTION set elem + >x> + >n.*
  The n-th bit in x is made equal to 1. It is required that $0 \leqslant n <$ *word size.*
- *FUNCTION clear elem + >x> + >n.*
  The n-th bit in x is made equal to 0. It is required that $0 \leqslant n <$ *word size.*
- *FUNCTION extract bits + >x + >n + y>.*
  A word is yielded in y that contains copies of the right-most n bits in x in the corresponding positions, and 0s in the remaining positions, if any. It is required that $0 \leqslant n \leqslant$ *word size.*
- *QUESTION first true + >x + n>.*
  If x contains at least one 1, *first true* succeeds and yields the position of the left-most 1 in n. Otherwise it fails.

- *FUNCTION pack bool + from[] + >n + word>.*
  The right-most n bits of *word* are filled as follows. If the element in *from* with address $>>$*from* $- i$ contains at least one 1, bit i of *word* is set to 1, and otherwise to 0, for $0 \leqslant i < n$. The remaining bits in *word*, if any, are 0. It is required that $0 \leqslant n <$ *word size.*
- *ACTION unpack bool + >word + []st[].*
  The stack *st* is extended with *word size* blocks of one location each, the location with address $>>st - i$ containing a copy of the i-th bit in *word*, for $0 \leqslant i <$ *wordsize.*

5.2.3. Strings

For those data that are considered to be strings and characters the following externals are available.

- *CONSTANT max char.*
  max char has the maximum integer value that corresponds to a character.
- *FUNCTION to ascii + >c + d>.*
  d is given the integer value that corresponds in ASCII-code to the character that corresponds to *c* in the code used. It is required that $0 \leqslant c \leqslant$ *max char*.
- *FUNCTION from ascii + >c + d>.*
  d is given the integer value that corresponds in the code used to the character that corresponds to *c* in ASCII. It is required that $0 \leqslant c \leqslant 127$.
- *ACTION pack string + from[] + >n + []to[].*
  The right-most *n* elements of *from* must be values that correspond to characters. These characters are packed, in some way, into some number *m* of values, and the stack *to* is extended with *m* blocks of one location each, containing these values. The packed format thus obtained is the same as that used for storing strings in lists (see 4.1.5). The 'pointer' to the string is the address of the right-most element. So, after a call of *pack string*, the limit *>>to* is the pointer to the resulting packed string.
- *ACTION unpack string + from[] + >p + []to[].*
  The pointer *p* must point into the list *from* and be the address of a packed string. This string is unpacked yielding a sequence of *m* character values, and the stack *to* is extended with *m* blocks of one location each, containing these values in left-to-right order.
- *QUESTION string elem + text[] + >p + >n + c>.*
  The pointer *p* must point into *text* and be the address of a packed string. If this string has an *n*-th character (counting from 0), its value is yielded in *c* and *string elem* succeeds; otherwise it fails.
- *FUNCTION string length + text[] + >p + n>.*
  The pointer *p* must point into *text* and be the address of a packed string. The number of characters in this string is yielded in *n*.
- *FUNCTION compare string + t1[] + >p1 + t2[] + >p2 + trit>.*
  The pointer *p1* must point into *t1* and be the address of a packed string, s1. The pointer *p2* must point into *t2* and be the address of a packed string, s2. These two strings are compared in some way: if s1 is smaller than (lexicografically comes before) s2, *trit* is set to -1; if they are equal, *trit* is set to 0; otherwise *trit* is set to 1.
- *ACTION unstack string + []st[].*
  The 'max limit' of *st* must point into *st* and be the address of a packed string. The blocks containing this string are removed from *st*.
- *ACTION previous string + t[] + >pnt>.*
  The pointer *pnt* must point into *t* and be the address of a packed string; it is made to point to the (possibly non-existing) block just preceding the string.
- *QUESTION may be string pointer + text[] + >p.*
  Succeeds if *p* points into *text* and can be interpreted as the address of a packed string. Otherwise it fails.

5.2.4. Lists

For lists the following externals are available.

- *CONSTANT nil.*
  nil is a value that points into the standard table *nil table.*
- *TABLE nil table.*
  Contains one entry, *nil*, pointed at by *nil.*
- *QUESTION was + ()a[] + >p.*
  Succeeds if *p* points into *a*, fails otherwise.
- *FUNCTION next + ()a[] + >p>.*
  The calibre of *a* is added to *p.*
- *FUNCTION previous + ()a[] + >p>.*
  The calibre of *a* is subtracted from *p.*
- *FUNCTION list length + ()a[] + l>.*
  The number of elements in *a* is yielded in *l.*
- *ACTION unstack + [][]st[].*
  The stack *st* must contain at least one block. The right-most block of *st* is removed. Its locations can be reclaimed by an extension, its contents are lost.
- *ACTION unstack to + [][]st[] + >pnt.*
  Zero or more blocks are removed from the right hand side of *st*, so that the 'max limit' of *st* becomes equal to *pnt*. If this cannot be done, an error message follows.
- *ACTION unqueue + [][]st[].*
  The stack *st* must contain at least one block. The left-most block of *st* is removed. Its (virtual) locations and its contents are lost.
- *ACTION unqueue to + [][]st[] + >pnt.*
  Zero or more blocks are removed from the left hand side of *st*, so that the 'min limit' of *st* becomes equal to *pnt*. If this cannot be done, an error message follows.
- *ACTION scratch + [][]st[].*
  All blocks in *st* are removed. Their locations can be reclaimed through extensions, their contents are lost.
- *ACTION delete + [][]st[].*
  All blocks in *st* are removed, as in a call of *scratch.* Moreover, the run-time system will disregard *st* until a possible subsequent extension on *st.* Consequently, the remaining stacks may get better service, but reactivating *st* may be expensive.

5.2.5. Files

The following standard externals on files are available.

- *CONSTANT new line, same line, new page.*
  These constants are predefined values to be used as control integers for 'charfiles'. Their intended meanings are 'print on new line', 'print again on same line' and 'print on first line of next page' respectively, as far as meaningful for the charfile and as far as implementable in the system.

- *CONSTANT rest line.*
  rest line acts as a dummy control integer and is used by *get line*, *put line* and *put char*.
- *PREDICATE get line + ""file + []st[] + cint>.*
  The file *file* must be a charfile. If the file is exhausted, *get line* fails. Otherwise the next item in *file* is read; if it is a control integer, it is assigned to *cint*, otherwise *cint* is set to *rest line*. Then zero or more characters are read from *file* until the end of the line. The stack *st* is extended with these characters in left-to-right order.
- *ACTION put line + ""file + a[] + >cint.*
  The file *file* must be a charfile; *a* must only contain values that correspond to characters. If *cint* is not *rest line*, a line with control integer *cint* is written on file *file*, containing the characters in *a* in left-to-right order. Otherwise the characters in *a* are appended to the last line written on *file*.
- *PREDICATE get char + ""file + char>.*
  The file *file* must be a charfile. If the file is not exhausted, the next character or control integer is read and delivered in *char*. Otherwise *get char* fails.
- *ACTION put char + ""file + >char.*
  The file *file* must be a charfile. The value of *char* must either correspond to a character or be a control integer. This character or control integer is written on file *file*, except the control integer *rest line*, which is ignored.
- *ACTION put string + ""file + text[] + >p.*
  The file *file* must be a charfile; the pointer *p* must point into *text* and be the address of a packed string. This string is written on the file *file*.
- *PREDICATE get int + ""file + int>.*
  The file *file* must be a charfile. A call of *get int* will read and skip any number of spaces and control integers on *file* until it either reaches the end of the file, in which case it fails, or finds a digit, plus-sign or minus-sign. It will then read and collect one or more digits until a non-digit is found: this non-digit is not read. The value of this stream of digits considered as a signed decimal number is given in *int*.
  A subsequent call of *get char* will yield the non-digit mentioned. If the above cannot be performed, an error message is given.
  This rule involves backtrack. It is not intended for use in programs that handle input very carefully; it is meant to provide an easy means for reading numbers.
- *ACTION put int + ""file + >int.*
  intsize + 1 characters are appended to the last line on *file*, which must be a charfile. These characters are: zero or more spaces, the sign of *int* and the characters of the decimal representation of the absolute value of *int* without leading zeroes.
- *CONSTANT numerical, pointer.*
  These constants are predefined values that can be used as type indications in datafiles. For their meanings see 4.2.2.
- *PREDICATE get data + ""file + data> + type>.*
  The file *file* must be a datafile. If the file is not exhausted, the next data-item is read, its value delivered in *data* and its type in *type*. Otherwise it fails. For a more detailed description see 4.2.2.

- *ACTION put data + ""file + >data + >type.*
  The file *file* must be a datafile. A data-item is written on the file, consisting of the value *data* and the type *type*. For a more detailed description see 4.2.2.
- *PREDICATE back file + ""file.*
  If there is not yet a last item read, *back file* fails. Otherwise it succeeds and the file is repositioned to beginning of the file.

6. PRAGMATS

Pragmats are used to control certain aspects of the compilation ('compiler-pragmats') and to supply implementation-dependent information to the machine-dependent part of the compiler ('user-pragmats'). The exact position of a compiler pragmat in the program may be significant.

Syntax:

pragmat:
    pragmat symbol, pragmat item list, point symbol.
pragmat item list:
    pragmat item, [comma symbol, pragmat item list].
pragmat item:
    tag;
    tag, equals symbol, pragmat value;
    tag, equals symbol, pragmat value list pack.
pragmat value:
    tag;
    integral denotation;
    string denotation.
pragmat value list pack:
    open symbol, pragmat value list, close symbol.
pragmat value list:
    pragmat value, [comma symbol, pragmat value list].

Example:

PRAGMAT title = "aleph compiler",
    background = (numb adm, history),
    macro = (convert 1 to 2 compl, set all bits).

Before the meaning of a **pragmat** is determined, it is preprocessed: all **pragmat-value-list-packs** are removed in the following way.

For every **pragmat-value-list-pack** which is preceded by an **equals-symbol** preceded by a **tag**, the **equals-symbol** and **tag** are removed and inserted in front of each **pragmat-value** in the **pragmat-value-list-pack**.

Subsequently all **open-symbols** and **close-symbols** are removed.

Thus the **pragmat-item** *background = (numb adm, history)* has the same meaning as

background = numb adm, background = history,

All **pragmat-items** now consist either of a single **tag** or of **tag, equals-symbol** followed by a **tag, integral-denotation** or **string-denotation**. They are divided into two groups according to the first **tag**: 'compiler-pragmats', affecting the compiler and 'user-pragmats'.

6.1. Compiler-pragmats

The **tags** *background, compile, count, dump, first col, last col, macro* and *title* identify "compiler-pragmats".

- *background* = **list-tag**
  The identified list will be kept on background memory if possible and necessary. The position of this **pragmat** is immaterial.
- *compile* = **tag**
  The **tag** can be:
  off: subsequent program text will be interpreted in the following sense:
  a. the **rule-body** of a **rule-declaration**, the **rule-tag** of which is used in normally compiled text will be interpreted as dummy,
  b. a **rule-declaration** the **rule-tag** of which is not used in normally compiled text will be ignored,
  c. a **data-declaration** will be ignored,
  d. a **pragmat-item** other than *compile = on* will be ignored.
  Injudicious application of this pragmat can render a correct program incorrect.
  on: normal compilation is resumed.
  all: subsequent **pragmat-items** of the form *compile = off* will have no effect.
  The standard option is *on*.
- *count* = **tag**
  The **tag** can be:
  rule: a counter is kept for each subsequent rule and compound member. The initial value of the counter is 0; it is incremented by 1 for every entrance to its rule or compound member. The counters are printed at program termination.
  member: same as for rule, except that a counter is kept for every member.
  off: no counters are kept for subsequent program text.
  The standard option is *off*.
- *dump* = **tag**
  The **tag** can be:
  global: upon error termination a symbolic dump of all global variables and stacks will be printed.
  rule: upon error termination a symbolic dump of the run-time stack will be printed.

member: upon error termination the number of the current member (as determined by the compiler) will be printed.

The position of this pragmat in the program is immaterial. The standard option is *member*.

- *first col* = **integral-denotation**
  Call the value of the **integral-denotation** i. The first $i-1$ characters on subsequent program lines are ignored. This alignment can be revoked in another *first col* pragmat. An initial pragmat *first col* = *1* is assumed.
- *last col* = **integral-denotation**
  Call the value of the **integral-denotation** i. All characters beyond the i-th position on subsequent program lines are ignored. This alignment can be revoked in another *last col* pragmat. An initial pragmat *last col* = *72* is assumed.
- *macro* = **rule-tag**
  The **rule-tag** must identify a non-recursive rule. Calls of this rule will be implemented through textual substitution rather than by subroutine call. The **rule-tag** may not be the **rule-tag** of the **affix-form** of the **root**. This pragmat must occur before the declaration of the affected rule.
- *title* = **string-denotation**
  The **string-denotation** is the title of the program. The default title is empty.

6.2. External-pragmats

Deleted.

6.3. User-pragmats

Pragmats not identified in 6.1 are considered "user-pragmats" and are transferred to the implementation-dependent part of the compiler.

7. THE REPRESENTATION OF PROGRAMS

7.1. The program

The program produced by the notion **program** consists of a series of terminal symbols. Into this program comments may be inserted in the following way.

The program is considered as a sequence of the following units:

tags,
integral-denotations,
character-denotations,
string-denotations and
symbols not occurring in one of the above.

Spaces may be added in front of all these units and inside **tag**s and **integral-denotation**s.

Long comments may be added in front of all these units. A long comment consists of a dollar-sign ($), followed by zero or more characters which are not dollar-signs, followed by a dollar-sign.

Short comments may be added in front of all units except **tag**s and **integral-denotation**s. A short comment consists of a sharp-sign (#) followed by zero or more

letters, digits and spaces.

In the program thus obtained all symbols are expanded into characters as described in 7.2 (e.g., **root-symbol** turns into *ROOT*).

The program text is then divided into lines in such a way that no comment is spread over two or more lines. If a line ends with a dollar-sign from a long comment, this dollar-sign may be omitted. In other words: long comments start with a dollar-sign and end at a dollar-sign or at the end of the line; short comments start with a sharp-sign and end at the first character which is not a letter, a digit or a space, or at the end of the line.

Depending on the pragmats *first col* and *last col* (see 6.1) a number of characters must be added before each line or may be added behind each line.

7.2. The characters

Almost all terminal symbols of the ALEPH grammar are notions that end in **-symbol**. The exceptions are **tag**, **digit**, **character** and **non-quote-item**. A **tag** is represented by a non-empty sequence of small letters and/or digits, the first of which is a small letter; two **tag**s are equal if their representations consist of equal sequences. A **digit** is represented by one of the digits *0* ... *9*. A **character** is represented by any character in the available character set except the new-line control character. A **non-quote-item** has as its representation any representation of **character** with the exception of the representation of the **quote-symbol**.

The representations of the other terminal symbols can be found in the following table.

symbol	*representation*
absolute-symbol	/
action-symbol	*ACTION* or *ACT*
actual-affix-symbol	+
box-symbol	=
bus-symbol	]
by-symbol	/
charfile-symbol	*CHARFILE*
close-symbol	)
colon-symbol	:
comma-symbol	,
constant-symbol	*CONSTANT* or *CST*
datafile-symbol	*DATAFILE*
dummy-symbol	?
end-symbol	*END*
equals-symbol	=
exit-symbol	*EXIT*
external-symbol	*EXTERNAL*
failure-symbol	−
formal-affix-symbol	+
function-symbol	*FUNCTION* or *FCT*
left-symbol	<
local-affix-symbol	−
minus-symbol	−

of-symbol	*
open-symbol	(
plus-symbol	+
point-symbol	.
pragmat-symbol	*PRAGMAT*
predicate-symbol	*PREDICATE* or *PRED*
question-symbol	*QUESTION* or *QU*
quote-symbol	"
repeat-symbol	:
right-symbol	>
root-symbol	*ROOT*
semicolon-symbol	;
stack-symbol	*STACK*
sub-symbol	[
success-symbol	+
table-symbol	*TABLE*
times-symbol	*
up-to-symbol	:
variable-symbol	*VARIABLE* or *VAR*

8. EXAMPLES

8.1. Towers of Hanoi

```
$ Towers of Hanoi.
CHARFILE print = "output">.

ACTION move tower + >length + >from + >via + >to:
    length = 0;
    decr + length, move tower + length + from + to + via,
        move disc + from + to,
        move tower + length + via + from + to.

ACTION move disc + >s1 + >s2:
    put char + print + s1, put char + print + s2,
        put char + print + / /.

ROOT move tower + 6 + /a/ + /b/ + /c/.

END
```

8.2. Printing Towers of Hanoi

```
$ Towers of Hanoi, full printing of the towers.
CHARFILE print = "output">.
```

```
STACK [1] a, [1] b, [1] c.

CONSTANT size = 5.

ACTION move tower + >length + []from[] + []via[] + []to[]:
    length = 0;
    decr + length, move tower + length + from + to + via,
        move disc + from + to, print towers,
        move tower + length + via + from + to.

ACTION move disc + []st1[] + []st2[]:
    * st1[>>st1] → st2 * st2, unstack + st1.

ACTION print towers − ln:
    size → ln,
    (lines:
        ln = 0;
        print disc + a + ln, print disc + b + ln,
            print disc + c + ln, put char + print + new line,
            decr + ln, :lines
    ).

ACTION print disc + []st[] + >line − index:
    minus + line + 1 + index, plus + index + <<st + index,
        (   was + st + index, print actual disc + st[index];
            print blank disc
        ).

ACTION print actual disc + >nmb − spc:
    minus + size + nmb + spc,
        repeat + spc + / /, repeat + nmb + /*/, repeat + 1 + /*/,
        repeat + nmb + /*/, repeat + spc + / /.

ACTION print blank disc:
    repeat + size + / /, repeat + 1 + / /, repeat + size + / /.

ACTION repeat + >cnt + >sb:
    cnt = 0;
    put char + print + sb, decr + cnt, :repeat.

ACTION play towers − n:
    size → n,
        (fill a: n = 0; decr + n, * n→a * a, :fill a),
        print towers, move tower + size + a + b + c.
```

```
ROOT play towers.

END
```

8.3. Symbolic differentiation

```
$ Symbolic differentiation, problem III in 'Machine Oriented
$ Languages Bulletin', MOLB 3.1.2, 1973.

CHARFILE out = "output">.

STACK [100] (op, left, right) expr.

TABLE operator =
        ("+": plus op, "−": min op, "*": tim op, "/": div op,
        "ln": ln op $ ln(f) is represented as 0 "ln" f $,
        "pow": pow op $ pow(f, g) is represented as f "pow" g $).

STACK [10] const = (0: c zero, 1: c one, 2: c two).

STACK [1] var = ("x": x var).

ACTION derivative + >e + de> − f − df − g − dg − n1 − n2 − n3:
    was + const + e, c zero → de;
    was + var + e, c one → de;
    left*expr[e] → f, right*expr[e] → g,
        derivative + f + df, derivative + g + dg,
        (    = op*expr[e] =
            [plus op], gen node + plus op + df + dg + de;
            [min op], gen node + min op + df + dg + de;
            [tim op],
                gen node + tim op + f + dg + n1,
                gen node + tim op + df + g + n2,
                gen node + plus op + n1 + n2 + de;
            [div op],
                gen node + tim op + df + g + n1,
                gen node + tim op + f + dg + n2,
                gen node + min op + n1 + n2 + n1,
                gen node + pow op + g + c two + n2,
                gen node + div op + n1 + n2 + de;
            [ln op], gen node + div op + dg + g + de;
            [pow op],
                gen node + min op + g + c one + n1,
                gen node + pow op + f + n1 + n1,
                gen node + tim op + df + g + n2,
                gen node + tim op + n2 + n1 + n1,
                gen node + ln op + c zero + f + n2,
                gen node + tim op + n2 + dg + n2,
                gen node + pow op + f + g + n3,
```

```
                gen node + tim op + n2 + n3 + n2,
                gen node + plus op + n1 + n2 + de
            ).

ACTION print expr + >e - zz:
    was + const + e, put int + out + const[e];
    was + var + e, put string + out + var + e;
    op*expr[e] → zz,
        (   = zz =
            [plus op; min op; tim op; div op],
                put char + out + /(/,
                print expr + left *expr[e],
                put char + out + /)/,
                put string + out + operator + zz,
                put char + out + /(/,
                print expr + right*expr[e],
                put char + out + /)/;
            put string + out + operator + zz, put char + out + /(/,
                (   equal + zz + pow op,
                        print expr + left*expr[e],
                        put char + out + /,/;
                    +
                ),
                print expr + right*expr[e], put char + out + /)/
        ).

ACTION test - e1 - e2 - e3:
    gen node + pow op + x var + x var + e1, $ pow(x, x)
        print expr + e1, nl,
    derivative + e1 + e2, print expr + e2, nl,
    derivative + e2 + e3, print expr + e3, nl,
    gen node + div op + x var + x var + e1, $ x/x
        print expr + e1, nl,
    derivative + e1 + e2, print expr + e2, nl,
    derivative + e2 + e3, print expr + e3, nl.

ACTION gen node + >op + >left + >right + res>:
    * op → op, left → left, right → right * expr,
        >>expr → res.

ACTION nl: put char + out + new line.

ROOT test.

END
```

8.4. Quicksort

```
ACTION quicksort + >from + >to + []a[]
                    - left - middle - right - a middle:
$
$ This rule sorts the elements in the stack 'a' from 'from' to
$ 'to' in ascending order. The algorithm used is a variation of
$ 'quicksort', C.A.R. Hoare, Computer J. 5, 10-15, 1962.
$
    mreq + from + to;
    $ The area to be sorted is not empty;
    $ it is split into three parts: left, middle and right.
    $ The middle contains one or more equal elements.
    from → left, random + from + to + middle, to → right,
        a[middle] → a middle,
        (split:
            (push right:
                more + left + to;
                more + a[left] + a middle;
                incr + left, : push right
            ),
            (push left:
                more + from + right;
                more + a middle + a[right];
                decr + right, : push left
            ),
            (less + left + right,
                ( - elem:
                a[left] → elem, a[right] → a[left], elem → a[right]
                ),
                incr + left, decr + right, : split;
            less + middle + right,
                a[right] → a[middle], a middle → a[right],
                decr + right;
            more + middle + left,
                a[left] → a[middle], a middle → a[left], incr + left;
            +)
        ),
        quicksort + from + right + a, quicksort + left + to + a.
```

8.5. Permutations

```
$ 'next perm' considers the right-most 'n' elements of 'st'
$ as a permutation and replaces them by the elements of the next
$ permutation in lexicographical order. If there is no next
$ permutation, 'next perm' fails.
```

```
PREDICATE next perm + >i + []st[] - p:
    less + i + >>st, plus + i + 1 + p,
        (   next perm + p + st;
            less + st[i] + st[p], simple perm + i + st
        ).

ACTION simple perm + >i + []st[] - p - q:
    $ the right-most 'i' elements of 'st' do have a next permutation,
    $ but the right-most 'i-1' don't.

    >>st → q,
        (find new ith elem:
            less + st[q] + st[i], decr + q, :find new ith elem;
            +
        ), swap + st + i + q,
        plus + i + 1 + p, >>st → q,
        (invert perm tail:
            mreq + p + q;
            swap + st + p + q, incr + p, decr + q, :invert perm tail
        ).

ACTION swap + []st[] + >i1 + >i2 - elem:
    st[i1] → elem, st[i2] → st[i1], elem → st[i2].

STACK st = (/1/, /2/, /3/, /4/).

ROOT display perms + st.

ACTION display perms + []st[]:
    put line + output + st + new line,
        (next perm + <<st + st, :display perms; +).

CHARFILE output = "output">.

END
```

9. REFERENCES IN THE MANUAL

[BÖHM 77] A.P.W. Böhm, ALICE: An Exercise in Program Portability, IW 91/77, Mathematical Centre, Amsterdam, 1977.

[GLANDORF, GRUNE & VERHAGEN 78] R. Glandorf, D. Grune & J. Verhagen, A W-grammar of ALEPH, IW 100/78, Mathematical Centre, Amsterdam, 1978.

[KOSTER 71a] C.H.A. Koster, A Compiler Compiler, MR 127/71, Mathematical Centre, Amsterdam, 1971.

[KOSTER 71b] C.H.A. Koster, Affix Grammars, *in* J.E.L. Peck (Ed.), ALGOL 68 Implementation, North-Holland Publ. Co., Amsterdam, 1971.

[WATT 77] D.A. Watt, The Parsing Problem for Affix Grammars, Acta Inf. *8*, 1-20, 1977.

[WICHMANN 77] B.A. Wichmann, How to Call Procedures, or Second Thoughts on Ackermann's Function, Software — Practice & Experience *7*, 317-329, 1977.

10. INDEX

The main references are in **bold**, references to the syntax are in *italic*.

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z