AN EARLY IMPLEMENTATION OF REVISED ALGOL 68

garbage collection using recognizable pointers

Proefschrift

ter verkrijging van de graad van
doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus,

prof. drs. P. A. SCHENCK,

in het openbaar te verdedigen ten overstaan van
een commissie aangewezen door het college van Dekanen
op 14 November 1989
te 14.00 uur
door

Josephus Johannes Franciscus Maria Schlichting,

geboren te Amsterdam,
doctorandus in de Wiskunde.

Dit proefschrift is goedgekeurd door de promotor

prof. dr. ir. W.L. van der Poel.

aan Marianne

EEN VROEGE IMPLEMENTATIE VAN HERZIEN ALGOL 68

garbage collectie met herkenbare verwijzingen

SAMENVATTING

Dit proefschrift beschrijft enige aspecten van de ALGOL 68 compiler voor de Control Data 6000 en opvolger computers, die in de jaren 1972 tot 1985 in opdracht van de Universiteiten van Utrecht en Groningen en van de stichting SARA te Amsterdam werd geimplementeerd door Control Data B.V. onder leiding van de schrijver.

De implementatie werd begeleid door een stuurgroep, bestaande uit S.G. van der Meulen van de Universiteit van Utrecht, D. Grune van het Mathematisch Centrum te Amsterdam en de auteur. W.L. van der Poel trad op als adviseur van Control Data voor het project.

Het hoofddoel van het project was een zo getrouw mogelijke implementatie van het "Revised Report on the Algorithmic Language ALGOL 68". Als uitbreidingen zijn o.a. faciliteiten voor afzonderlijke compilatie van procedures en voor definitie van operatoren in termen van machine code toegevoegd.

De FORTRAN interface, een uitbreiding van ALGOL 68, die het mogelijk maakt in FORTRAN geschreven routines aan te sluiten, werd in opdracht van de Universiteit van Groningen toegevoegd naar specificaties die in samenwerking met C.G. van der Laan werden opgesteld. Hierbij kunnen niet alleen parameters van typen, die door ALGOL 68 en FORTRAN ondersteund worden, maar ook in ALGOL 68 geschreven procedures aan FORTRAN routines worden doorgegeven, zodat subroutine bibliotheken zoals NAG en IMSL vanuit ALGOL 68 volledig gebruikt kunnen worden.

De compiler omvat zes fasen, te weten de lexicale, syntactische en semantische analyse van het te compileren programma en de productie, optimisatie en redactie van de object code.

Het ontwerp van het runtime systeem en van de garbage collector in het bijzonder is gebaseerd op een zodanige interne voorstelling van gegevens, dat verwijzingen als zodanig herkenbaar zijn. Hierdoor werd het mogelijk een snelle en compacte garbage collector te realiseren.

CURRICULUM VITAE

J.J.F.M. Schlichting

Geboren 29 Januari 1933 te Amsterdam.
Bezocht het St. Ignatius College te Amsterdam en deed in 1951 eindexamen Gymnasium B.
Studeerde aan de Gemeentelijke Universiteit te Amsterdam.
Vervulde in de jaren 1955 - 1957 de militaire dienstplicht.
Was part-time assistent bij het Mathematisch Centrum te Amsterdam in de jaren 1958 - 1960.
Trad in dienst van Univac Nederland in 1962.
Deed doctoraal examen Wiskunde in 1965.
Werkte in dienst van Auerbach Nederland in 1967.
Van af 1969 werkzaam bij Control Data B.V.

PREFACE.

This thesis describes some aspects of an implementation of ALGOL 68 that was undertaken in the years 1972 to 1985 by Control Data B.V. under the project-leadership of the author. This project was part of a contract of Control Data and three of its Dutch users, the Universities of Groningen and Utrecht and SARA, a foundation of the Mathematical Centre, the University of Amsterdam and the Free University at Amsterdam. In 1975 the first version of the compiler was delivered. At a later time a FORTRAN interface was implemented under a contract with the University of Groningen.

A steering group consisting of S.G. van der Meulen from the University of Utrecht, D. Grune from the Mathematical Centre and the author supervised the project; W.L. van der Poel acted as external advisor to Control Data for the project and D. Gilinsky of Computer Sciences Corporation acted as advisor to the implementation team.

The major assignments were as follows:

External specifications:	Steering Group.
Project management:	J. Schlichting.
Global design:	J. Schlichting and D.Gilinsky.
Lexical scan:	T. de Vries.
Syntactic scan:	P. van Oostrum and T. Zoethout.
Semantic scan:	H. van Hedel and D. van Ligten.
Code generator:	J. Schlichting and P. van Oostrum.
Code optimizer:	J. Anderson and J. Schlichting.
Editor:	J. Anderson.
Run time organization:	J. Schlichting.
Transput:	T. Zoethout and D. van Ligten.

FORTRAN interface design: J.Schlichting.

Although the contract called for an implementation of the original Report on the Algorithmic Language ALGOL 68, it was decided to follow the development of the Revised Report as closely as possible.

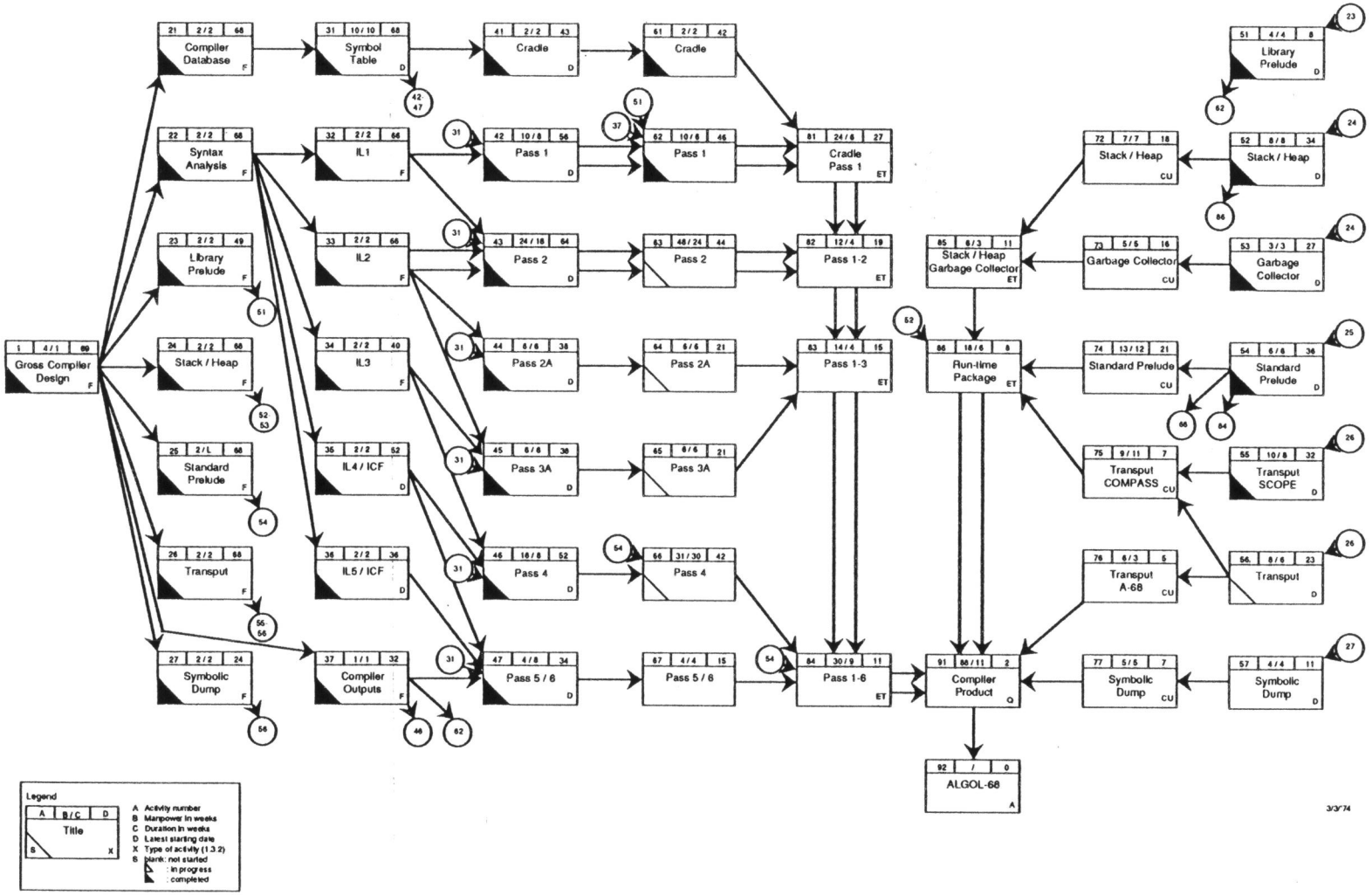
The involvement of W.L. van der Poel, S.G. van der Meulen, both members of the defining committee, IFIP WG 2.1, and D. Grune of the Mathematical Centre, represented by three persons in WG 2.1., ensured the availability of accurate and up to date information on the language being developed.

The primary aim of the project was to comply as completely as possible with the Revised Report in the sense of accepting all programs valid according to the Revised Report and possibly some others. Eventually the compiler delivered in June 1975 deviated only slightly from the Report which had not yet been published at that time.

The specification of the FORTRAN interface was established in close cooperation with C.G. van der Laan of the University of Groningen. It was designed to facilitate the use of available

FORTRAN routine libraries such as NAG and IMSL with the ALGOL 68 product.

A detailed project plan was established. The next page shows an overview of the project in the form of a PERT-chart.


Gross Compiler Design
Compiler Database
Symbol Table
Cradle
Cradle
Syntax Analysis
IL1
Pass 1
Pass 1
Cradle Pass 1
Library Prelude
Stack / Heap
IL2
Pass 2
Pass 2
Pass 1-2
Stack / Heap Garbage Collector
Garbage Collector
Stack / Heap
Standard Prelude
IL3
Pass 2A
Pass 2A
Pass 1-3
Run-time Package
Standard Prelude
Transput
IL4 / ICF
Pass 3A
Pass 3A
Transput COMPASS
Transput SCOPE
Symbolic Dump
IL5 / ICF
Pass 4
Pass 4
Transput A-68
Transput
Compiler Outputs
Pass 5 / 6
Pass 5 / 6
Pass 1-6
Compiler Product
Symbolic Dump
Symbolic Dump
ALGOL-68
Legend
Title
A Activity number
B Manpower in weeks
C Duration in weeks
D Latest starting date
X Type of activity (1,3,2)
S blank: not started
: in progress
: completed
3/3/74

ACKNOWLEDGEMENTS

I am grateful to Control Data, for entrusting me with the implementation of ALGOL 68 and enabling me to write this thesis.

I am grateful to S.G. van der Meulen and D.Grune, members of the steering group, for their guidance and their positive attitude in negotiating compromises.

My gratitude also concerns C.G. van der Laan for the cooperation in the definition of the FORTRAN interface.

Not in the last place I want to express my gratitude to the members of the implementation team, whose commitment and perseverance were instrumental in the success of the project.

TABLE of CONTENTS

1 INTRODUCTION.

This thesis describes some aspects of an implementation of ALGOL 68 realized under the project leadership of the author.

The external specification of the implementation may be found in "ALGOL 68 Version II Reference Manual" [25]. An overview of the machine independent part of the compiler is presented in "De Cyber ALGOL 68 vertaler" [24] by the author.

1.1 The Language to be Implemented.

When the Control Data ALGOL 68 project was started in September 1972, the Report on the Algorithmic Language ALGOL 68 [1] had been available for nearly four years. The language was still being revised however by Working Group 2.1 on Programming Languages of the International Federation of Information Processing.

Between August 1972 and September 1973 five intermediate versions were issued by WG.2.1. In the sequel these versions will be referred to as Intermediate Report 1, 2, 3, 4 and 5 or IR1, IR2, IR3, IR4 and IR5 for short. IR1 [3], including only the syntax of the language, was issued in August 1972, followed by IR2 [4] in February 1973, IR3 [5] in April 1973, IR4 [6] in July 1973 and IR5 [7] in September 1973.

Four reports on improvements considered were published in the ALGOL Bulletin under the title: "The Report on improvements to ALGOL 68". The four issues appeared in AB 34.3.2 [8], AB 34.3.3 [9], AB 35.3.1 [10] and AB 36.1 [11].

The Revised Report on the Algorithmic Language 68 [2] was finally issued in 1974. In the sequel we will use the abbreviation "RA68" to refer to this report.

In IR2 a new method of description using a two-level grammar was introduced. Two level grammars, or "van-Wijngaarden-grammars", are explained in "Grammars for Programming Languages" [42].

Most design issues were resolved in IR4. The differences between the original language and the revised language are described in: R. Uzgalis, Language Changes incorporated in the Revised ALGOL 68 Report, draft dated July 1973 [12].

Since we were committed to implement RA68 as completely as possible within the constraints of time and resources available for the project, and RA68 had yet to appear, we attempted to make the design of compiler and run-time system sufficiently general and flexible to allow incorporation of changes to the language into the product under construction.

Examples of generalization are the use of a parser generator to produce the kernel of the syntactic analysis and the production

of a specialized macro processor to automate the code generation for the ALGOL 68 operators defined in the standard prelude.

Examples of flexibility are the implementation of two separate scans for lexical and syntactic analysis of the program text and the allocation on the heap of local generators.

In a few cases we had to decide on the incorporation of certain language features before the matter was settled in WG 2.1. e.g. the alternate representations of the sub- and bus-symbols.

1.2 Target Machines.

The target machine was the Control Data 6000, Cyber 170 [52], Cyber 700 ,Cyber 800 and Cyber 180 series of computers under the NOS [44, 45] and NOS/BE [46] operating systems.

The compiler can also be used on the Control Data 7000 series machines under the SCOPE 2 operating system [47].

Eventually a special version of the compiler was developed that allowed to exploit the architectural features of the 7000 series.

1.3 Testing the Product.

The compiler was tested before delivery using "The Revised MC ALGOL 68 Test set", edited by D. Grune [29].

1.4 Notational conventions.

Syntax rules are given in BNF with the addition of the meta rules:

{<entity>} ::= | <entity>.

{<entity>} ::= | (<entity) {<entity>} .

ALGOL 68 source text is presented in upper stropping and the comment-symbol is denoted by "#".

Arbitrary modes are designated by "AMODE", "BMODE", etc.

1.5 Contents of Thesis.

In the second chapter the language actually implemented and the deviations from RA68 will be discussed.

The third chapter describes the compilation process. An overview of the six passes will be presented.

The fourth chapter describes the design of the run-time system. The emphasis will be on the design and implementation of the garbage collector.The implementation of parallel clauses will be described in some detail.

The fifth chapter contains a summary, an analysis of the applicability in a wider scope of some design concepts used in this implementation and a discussion of some aspects of this implementation in the light of available literature.

The sixth chapter lists the references.

2 LANGUAGE IMPLEMENTED.

In this chapter we discuss the language actually implemented.

2.1 Between Report and Revised Report.

Because the project was started before the Revised Report appeared we had to decide on the implementation of certain language features before they were definitely settled in WG.2.1.

2.1.1 Alternate representations of sub- and bus-symbols.

The alternate representations of the sub- and bus-symbols were not yet defined when the project started. We decided to use "(/" and "/)" rather than "(" and ")" , mainly because it leads to a simplification of the parser.

2.1.2 Flexible multiples.

The concept of flexible multiples was subject to frequent and considerable changes. The main issues here were :

- is flexibility a property of the mode, of a bound of a multiple or of the multiple as a whole ?
- will an empty flexible multiple contain a ghost element i.e. an element containing only the bound information for its components ?
- which restrictions apply to transient names i.e. references to components of a flexible multiple ?

These issues will be discussed in the following three subsections.

2.1.2.1 Flexibility.

In ALGOL 68 we distinguish fixed and flexible multiples. The bounds of a fixed multiple are established once and for all when the multiple is declared or generated. The bounds of a flexible multiple likewise are established when the multiple is declared or generated, but to a reference to a flexible multiple we may assign a multiple value with bounds different from the bounds of the multiple value referred to by the reference.

In the original report flexibility is a property of a (lower or upper) bound in an actual or formal multiple; accordingly a descriptor of a n-dimensional multiple contains 2n states, one for each bound, all of which may be 0 or 1, encoding a fixed or flexible lower or upper bound in the each of the n dimensions.

Already in IR1 flexibility is a property of the multiple value as a whole. Consequently a descriptor has a single state, which governs the flexibility of all 2n bounds of the multiple.

In IR1 the flexibility of a multiple value referred to by a name yielded by an actual declarer is known at compile-time, but formal declarers yield names that may refer to a fixed multiple or to a flexible multiple, as shown in the following examples:

actual declarers:

```
         [1:9] REAL fxr;    #fixed#
FLEX     [1:9] REAL flr;    #flexible#
```

formal declarers:

```
         [ ] REAL refx,     #fixed actual#
FLEX     [ ] REAL refl,     #flexible actual#
EITHER   [ ] REAL rexl,     #fixed or flexible actual#
```

virtual declarers:

```
REF [ ] INT ri = IF b THEN LOC FLEX [1:3] INT    #fixed#
                      ELSE LOC      [1:3] INT    #flexible#
                 FI;
```

According to the semantics of assignation (section 5.2.1) in IR1 and IR2 the following is legal:

```
FLEX [1:2] [1:3] CHAR flexfix;
FLEX       [1:3] CHAR flex;
           [1:3] CHAR fix;
flexfix :=  (flex, fix);
```

Here flexfix [1] refers to a flexible multiple, while flexfix [2] refers to a fixed multiple. Informal reports on the discussions in WG 2.1.made us assume that this was not the intention of the editors.

Fortunately the whole issue of flexibility was simplified in IR4:

(a) flexibility is a property of the name.
(b) flexibility is part of the mode (of a name).
(c) the EITHER option is abolished.

Unfortunately the changes (a) and (b) came too late for incorporation in the implementation. Obviously we accepted (c).

The reasons for not implementing (b) were twofold: first, the code sequences for assignation of multiples had been completed and second, the internal representation of mode trees had been designed and deflexing was believed to be difficult to implement in the existing coercion routine.
The author some years later designed deflexing in the coercion routine, which proved relatively simple, but he lacked the

resources for its implementation.

Thus in the Control Data implementation flexibility is still a property of the multiple value. The FLEX symbol is ignored in formal and virtual declarers and an actual parameter corresponding to a formal multiple may be flexible or fixed.

2.1.2.2 Ghost element.

The ghost element concept was introduced in the language to remember the bounds of a fixed multiple being a component (of a struct being a component of) a flexible multiple. A flat flexible multiple contains a single ghost element instead of no elements at all; a multiple being a (component of a) ghost element contains a descriptor but no elements. E.g. after the declaration:

```
FLEX  [1:0]  [1:8] REAL flex_to_fix;
```

flex_to_fix contains a ghost element of mode "[1:8] REAL". Upon assignation to flex_to_fix of a multiple of multiples of REAL the ghost element is used to verify the bounds of the component multiple of the multiple assigned to flex_to_fix. E.g.

```
flex_to_fix := HEAP [1:2] [1:6] REAL;
```

would cause a run-time error, because of the inequality of the bound pairs "[1:6]" in the global generator and "[1:8]" of the ghost element.

The original report did not include the concept of ghost element because, upon the declaration of flex_to_fix the boundscripts are elaborated "a sufficient number of times" i.e. once for each component multiple of flex_to_fix, in this case zero times.

All Intermediate Reports, except IR4, included the ghost element concept, although it was called a dummy element in IR2 and IR3.

Our final design of multiples, based on IR4, does not include the ghost element. When the ghost element was reintroduced, we were faced with the question how to check the bounds of a component-multiple upon an assignation to a reference to a flat multiple of which the components possess a mode "showing" a multiple mode.

We adopted the simplest solution to the problem by forcing a multiple that is a component of a flexible multiple, to be flexible; this solution accepts all programs that are legal in RA68 and a few others, such as the example above and e.g.

```
FLEX  [1:6]  [1:8] REAL flex_to_fix;
flexfix [1] := (1.9, 2.8);
flexfix [2] := (1.0, 2.0, 3.0);
flexfix [3:6] := HEAP [1:4] [6:9] REAL;
```

2.1.2.3 Transient names.

Also related to the concept of flexibility is the concept of transient name. A transient name is a subname of a reference to a flexible multiple. As the designers of the language desired to avoid the use of the heap where possible it was deemed necessary to allow to reuse the memory occupied by the elements of a flexible multiple upon assignation to the flexible multiple. As a consequence the use of transient names had to be restricted in order to avoid the existence of references to no longer allocated data. Transient names must not be assigned, ascribed, rowed or displayed.

In the original report as well as in all Intermediate Reports up to IR3 these restrictions are formulated in the semantics of assignation, identity declaration, rowing and display. In IR4, IR5 and RA68 transient names are excluded in the syntax of those language constructs.

As in the Control Data implementation flexibility is not part of the mode, transient names cannot be identified syntactically and thus run-time checks incorporated in the object code would be required to detect violations of the restrictions on the use of transient names.

Also in RA68 transient names cannot be an operand of the so-called assigning operators: "+:=", "+=:", etc., because the transient name must not be ascribed to a formal parameter; we felt this to be an undesirable restriction.

We decided not to restrict the use of transient names at all but to make them unnecessary by not releasing the memory occupied by a flexible multiple when a new value is assigned to it.

This decision is in line with the treatment of the scope rule, which we discuss in the next section.

2.1.3 Scope rule.

The scope rule (RA68 2.1.1.3) in ALGOL 68 prohibits the assignation of a value to a reference when the value has an "older" scope than the reference. This rule is needed to avoid references to data that may be overwritten because it is out of scope.
In many cases the scope of a value cannot be determined at compile-time, largely because a reference constitutes a value in ALGOL 68.
Meertens [22] describes an algorithm for static or compile-time checking of scopes in ALGOL 68.

The need for the scope rule for references stems from the assumption that local variables are allocated on the run-time stack or, in ALGOL 68 terminology, that the yield of a local generator refers to a location on the run time stack.

This approach minimizes the use of the heap and thus the activity of the garbage collector.

The disadvantage is that in many cases run-time checks are needed to enforce the scope rule. Moreover the determination of the scope of references at compile-time complicates the compiler. Another undesirable consequence of the scope rule is that one cannot associate a non-global event-routine with a file, because the "book" of the file has global scope.

We therefore decided to allocate all local generators on the heap, thereby obviating the scope rule for references and multiples.

As an optimization we later identified all variables for which all applied occurrences are either the target of an assignation or the object of a dereference coercion and called them *optimized variables*. Optimized variables can be safely allocated on the run-time stack , because their use cannot cause scope rule violations.

The scope of routines cannot so easily be forced to be global.

The scope of a routine is defined as the innermost scope of the scopes of all entities referred to in the routine. Thus if the scope of all routines is treated as global, all data referred to from within a routine but not local to that routine must be treated as global and consequently in practice all run-time stack segments must be allocated on the heap.

We considered to take that approach, the suggestion of which is due to Van der Poel. While agreeing that such an approach is feasible and would e.g. greatly simplify the implementation of parallel clauses, we yet decided against it, mainly because run-time segments associated with routine activations would have to be allocated on the heap in their maximum size and parameter passing would be more complex.

In this implementation routines are the only values that require scope checking. As a scope violation on routines does not create ill-effects until the routine is called we decided to check the scope of a routine only when the routine is called.

Therefore the scope of a routine may have to be checked at run time and thus the internal representation of a routine must contain a scope field that uniquely represents the activation of the range being the scope of the routine.

Rather than maintaining an activation count for each scope range we use the one primitive action in the language that produces a unique result for every execution, which is the global generator.

The scope field in the internal representation of a routine contains a reference to a word in the heap that is the yield of a

global generator executed at the start of the activation of the range that is the scope of the routine being called.

2.2 Preludes and Postludes.

A prelude is associated with a postlude . A prelude with its associated postlude is called a *circumlude*. Two categories of circumludes are supported, system circumludes and user circumludes.

2.2.1 System circumludes.

System circumludes are provided by the ALGOL 68 system. The user selects the system circumlude by means of a parameter on the compilation-control statement. The appropriate parameter is described in Appendix B. Four system circumludes exist:

1. stand a rd circumlude: the default circumlude, providing standard modes, operators and procedures.

2. long circumlude: a circumlude that adds operations on values of mode LONG REAL and LONG COMPL.

3. icftra circumlude: a circumlude that adds all that is needed to compile the run-time system.

4. torbas circumlude: a circumlude that adds the Torrix system to the LONG circumlude. Torrix is a programming system for operations on vectors and matrices, described by Van der Meulen en Veldhorst [27].

2.2.2 User circumludes.

User written circumludes are compiled with a special parameter on the compilation-control statement; the circumlude may be used in subsequent compilations. Selection of user circumlude is effected by choices of parameters on the compilation-control statement. Appendix B contains a description of compilation-control statement parameters.

Unlike modules in some other programming languages e.g. Modula, Ada and Extended Pascal, circumludes cannot be used in combination. One circumlude only can be specified on the compilation-control statement.

It is possible however to produce a circumlude that contains the definitions of multiple preludes in a nested fashion. In order to make the definitions in two circumludes A and B available in a program, the user must first compile circumlude A and then compile circumlude B specifying A as circumlude producing a new circumlude, say AB.
In a program compiled with AB as circumlude the definitions contained in A and B are included. The definitions from A reside in an outer range however. Therefore an indicator defined in both A and B signifies the meaning attributed to it in circumlude B.

2.3 Hardware Representation.

Since at the time of completion of the compiler no standard hardware representation was defined, a Control Data hardware representation had to be invented. The standard hardware representation has been implemented at a later time. The Control Data representation was retained for reasons of compatibility.

2.3.1 Control Data hardware representation.

Both single and double stropping of bold tags are supported. In single stropping a bold tag is represented by the corresponding tag preceded by an apostrophe, in double stropping by the corresponding tag surrounded by apostrophes. E.g. in single stropping the begin-symbol is represented by : "'begin" not followed by a letter or digit, in double stropping by "'begin'".

Single or double stropping is selected by control statement parameter and pragmats are provided to switch back and forth between single and double stropping. Upper and lower case letters may be used interchangeably except in STRING- and CHAR-denotations.

2.3.2 Standard hardware representation.

The standard hardware representation is defined in : "The Report on the Standard Hardware Representation for ALGOL 68" [13]. The three stropping regimes, point, upper and res stropping, as well as all pragmats defined in [13] are fully supported.

2.3.3 Stropping regimes.

As many as five stropping regimes are supported in this implementation. The examples below show how a few frequently used sequences may be represented in each of those stropping regimes:

stropping regime	begin-symbol	begin-symbol if-symbol	begin-symbol identifier
single stropping	'begin	'begin'if	'begin iden
double stropping	'begin'	'begin''if'	'begin'iden
point stropping	.begin	.begin.if	.begin iden
upper stropping	BEGIN	BEGIN IF	BEGINiden
res stropping	begin	begin if	begin iden

The initial stropping regime may be selected by a parameter on the compilation-control statement. The selection of a stropping regime implies the selection of a hardware representation.

The obvious default for the parameter concerned would be single

stropping for the Control Data representation and point stropping for the Standard representation and since an extra parameter to choose between the two seemed a bit too much, we had a little problem.

The solution adopted is heuristic in nature and is based on the assumption that the casual user is using one of the hardware representations supported without being aware of the existence of the other. The default depends on the first stropping character encountered in the source text; if it is an apostrophe, we start in single stropping; if it is a point, we start in point stropping.

2.3.4 General images.

As an extension so-called *general images* are added to allow specification of any 12-bit character in the source text; general images are supported in conjunction with the standard hardware representation only. The character sets used are described in Appendix A.

A general image consists of an apostrophe, an open-symbol, a non-empty digit-sequence and a close-symbol; it represents the 12-bit character with the absolute value represented by the digit-sequence, e.g. "'(49)" represents the 12-bits character, of which the value is 49, i.e. "1".

A series of general images may be abbreviated by replacing any subsequence ")'(" by a comma. Thus "13" may be represented by : "'(49)'(51)" or by "'(49,51)".
Within general images typographical display features are not allowed.

2.3.5 Letter aleph.

The language does not define a representation for letter aleph. The original Report states that the programmer cannot provide such a representation. Implementations however that use ALGOL 68 to implement the standard prelude and transput, require one.

In identifiers and indicants letter aleph is accepted only when the first parameter on the compilation-control statement is "0". It may then be represented by "'/". As its internal code has the absolute value 127, it could also be represented by "'(127)".

2.4 Deviations from the Revised Report.

Summarizing, the deviations from RA68 are:

1. Style i sub- and bus-symbols are represented by "(/" and "/)', respectively, rather than by "(" and ")".

2. Short modes are not supported. The standard operators for values of mode LONG INT, LONG BITS and LONG BYTES are not provided. The internal representations of values of mode

{LONG} LONG PRIM, where PRIM is INT, BITS, BYTES or REAL are the same as for values of mode LONG PRIM.

3. Flexibility is a property of a multiple value, rather than part of a mode. Thus REF [ ] REAL and REF FLEX [ ] REAL designate the same mode.

4. A name referring to a component of a flexible multiple is treated as a name referring to a component of a fixed multiple. The concept transient name does not exist in this implementation.

5. A multiple that is a component of a flexible multiple is forced to be flexible. A flat flexible multiple does not contain a ghost element.

6. All generators are handled as global; a scope violation can occur only when a routine is called from outside of its scope.

7. The concepts of particular-prelude and particular-postlude are not implemented; instead a "standard-prelude" and "standard-postlude" analogous to "library-prelude" and "library- postlude" are supported.

8. Format patterns are not supported. The entire format is staticized at the time it is associated with a file. This is more in the spirit of RA68 than the "elaborate on use" method dictated by RA68.
The routines "associate" and "make conv" are not provided.

9. The routines "putbin" and "getbin" accept parameters of mode PROC (REF FILE) VOID in addition to the modes required by RA68.

2.5 Language Extensions.

The following extensions are described in the subsections below:

. Additional Environment Enquiries.
. ICF-macros.
. Separate compilation.
. FORTRAN interface.

2.5.1 Additional environment enquiries.

A few additional environment enquiries were implemented i.a.

central processor time,
current date,
current time of day,
currently available memory,
maximum memory that can be made available.

Most of these enquiries are available in frequently used languages for the target computers.

2.5.2 ICF-macros.

ICF-macros allow the definition of operators in terms of machine instructions rather than an ALGOL 68 routine-text.

An ICF-macro is included in an INLINE pragmat. The specification contains a number of parameterized ICF-instructions. ICF (Intermediate Code File) is the intermediase file, produced by the code-generator phase and read by the code-optimization phase; ICF-instructions are basically registerless machine code instructions.

An ICF-macro is used in an operation definition of the form:

```
OP <formal-part> operator = PR inline ICF-macro PR SKIP;
```

The ICF-macro consists of an ICF-macro header, followed by a number of ICF-macro instructions. The simplest form of the header is:

```
HEAD <count>,<length>
```

where <count> is an integer denoting the number of ICF-macro instructions in the macro and <length> is an integer denoting the length in words of the result. The header must be contained in a single source line.

An ICF macro instruction may assume the form:

```
{<result-nr>} <ICF-code> <operand>, <operand> .
```

<result-nr> designates the ordinal of the result word defined as the result of the ICF-instruction,
<ICF-code> designates the hardware or pseudo-instruction and
<operand> designates an operand of the instruction.

Each ICF instruction must be written on a separate source line.

e.g. the definition :

```
   OP (INT, INT) INT + = PR inline
   head  1,1
1  addx   /1,/2
   PR SKIP;
```

defines the operator "+" for arguments of mode INT.

In this example the ICF-header specifies one ICF-instruction and a one word result. The first field "1" in the ICF-instruction specifies that the result of the instruction constitutes the first word of the result, in this case, the whole result; the

second field "addx" specifies an integer add instruction and the operand fields "/1" and "/2" specify the first and second argument of the macro as operands of the instruction.

ICF-macros were invented mainly to simplify implementation of the standard circumlude.

More information on ICF-macros is given in Appendix D.

2.5.3 Separate compilation.

Separate compilation is controlled by the XDEF, FEDX and XREF pragmats.

To define a procedure that is callable from a separately compiled program or procedure the routine-text is preceded by an XDEF pragmat and followed by an FEDX pragmat.
Thus the routine-text is replaced by:

```
PR xdef {<external-name>} PR
  <routine-text>
PR fedx PR
```

<external-name> is a name acceptable to the Cyber loader.

E.g. the separate compilation unit:

```
BEGIN
  OP (COMPL z1,z2) COMPL + =
    PR xdef z;add PR
     (re OF z1 + re OF z2, im OF z1 + im OF z2)
    PR fedx PR;

  PROC zero = (REF [ ]REAL a) VOID :
    PR xdef clear PR
      (FOR i = LWB a TO UPB a DO a[i] := 0.0 OD)
    PR fedx PR;

  SKIP
END
```

creates two externally defined procedures "z;add" and "clear" which may be referred to in other compilation units using a XREF pragmat.

To refer to a separately compiled procedure the form:

```
PR xref <external-name> PR SKIP
```

is used in place of a routine text.

e.g. the declarations:

```
OP (COMPL z1,z2):COMPL + = PR xref z;add PR SKIP;
```

```
PROC clear = (REF [ ] REAL a) VOID : PR xref PR SKIP;
```

refer to the external procedures defined above.

An external procedure written in some other language may be referred to by a unit of the form:

```
PR xref <external-name> <external-name> PR SKIP
```

where the first external-name designates an interface routine that converts the calling sequence and calls the external procedure designated by the second external-name.

A single interface routine named "A68FTN" is provided which allows to call a FORTRAN routine; the parameters must all be of mode REF to mode. This routine is made obsolete by the implementation of the FORTRAN interface described below but has been retained for reasons of compatibility.

Typically the separate compilation facility is used in conjunction with a user circumlude.
The placement of the declaration of external procedures in a circumlude provides a certain level of mode checking for separately compiled procedures. During the separate compilation of a procedure the compiler verifies that no declaration in the circumlude associates the external name specified for that procedure with an external procedure possessing a mode different from the mode of the procedure being compiled.

2.5.4 FORTRAN interface.

The FORTRAN interface allows to specify external routines as compiled by FTN, the FORTRAN compiler for the 6000 and 7000 series and to call such routines from an ALGOL 68 routine.
FTN [51] implements FORTRAN 77 with some extensions.

2.5.4.1 Declaration of a FORTRAN routine.

An identifier is declared to posses a routine written in FORTRAN by means of the FORTRAN pragmat. In such an identity declaration the routine-denotation is replaced by a FORTRAN pragmat and a SKIP-symbol. The FORTRAN pragmat has the format:

```
PR fortran {<external-name>} {<length>} PR .
```

<external-name> is a name acceptable to the loader that defines the entry-point name of the FORTRAN routine; if no <external-name> is given, the entry-point name is formed by truncation of the identifier in the procedure declaration to seven characters.

<length> is an INT-denotation that defines the length of the result of a FORTRAN CHARACTER function.

2.5.4.2 Modes of parameters of a FORTRAN routine.

The correspondence between internal representations of the following ALGOL 68 PRIMitive MODes (*PRIMOD*) and FORTRAN types is listed in the table below:

ALGOL 68 PRIMOD	FORTRAN type
INT	INTEGER
REAL	REAL
LONG REAL	DOUBLE PRECISION
COMPL	COMPLEX
BOOL	LOGICAL

The internal representations of an ALGOL 68 CHAR and a FTN CHARACTER are different; FTN uses a 6-bit character set and A68 uses a 12-bit superset of ASCII.
Parameters of mode CHAR and [ ] CHAR are acceptable however. The value of mode CHAR is converted to the corresponding value in the 6-bit character set and the address of the converted value is passed to the FTN routine.
Parameters of mode REF CHAR or REF [ ] CHAR are not supported.

The following table shows the ALGOL 68 modes acceptable in a call to a FORTRAN routine, the corresponding FORTRAN types and whether the routine may yield a value of that ALGOL 68 mode.

ALGOL 68 MODE	FORTRAN type	as result
PRIMOD	corresponding type	yes
REF PRIMOD	corresponding type	no
CHAR	CHARACTER	yes
[ ] CHAR	CHARACTER ( )	yes
REF [ ].PRIMOD	corresponding type ( )	no

For non-REF parameters a copy is made of the value and the address of the copy is passed to the FORTRAN routine.
For REF parameters the address field of the value is passed to the FORTRAN routine.

2.5.4.3 Result mode of a FORTRAN routine.

As shown in the "as result" column in the table above, the acceptable result modes of a routine designated FORTRAN are PRIMOD, CHAR and [ ] CHAR; the corresponding FORTRAN types are the types corresponding to the PRIMOD, to CHARACTER*1 and to CHARACTER*n, where n equals the value of <length> in the pragmat.

2.5.4.4 Passing an ALGOL 68 routine to FORTRAN.

In addition to the above a parameter of type PROC( )PRIMOD can be passed to a FTN routine provided the type of its parameters are in the table. Thus one can pass an ALGOL 68 routine to an FTN routine; however the routine passed as parameter should not itself have a parameter of type PROC. This restriction was considered acceptable because procedures with procedural parameters are not used as parameters in the FORTRAN libraries that our users desired to use in conjunction with ALGOL 68 programs
Lifting the restriction would have considerably complicated the implementation of the FORTRAN interface, as will be explained in section 4.5.2.5.

In FORTRAN the bounds of a parameter array are defined in the called routine by means of a dimension statement. In an ALGOL 68 routine that is used as parameter in a call to an FTN routine and that has at least one parameter of mode ROWS, the bounds of such parameters must be defined by means of a so-called descriptor pragmat, written immediately following the routine- symbol in the procedure declaration.

The syntax of this pragmat is defined by the rules:

```
<descriptor-pragmat> ::=
   PR descriptor <array-descriptor> {,<array-descriptor>} PR
   <array-descriptor> ::=
            <identifier> <bound-pair>, { <bound-pair> }.
   <bound-pair> ::= [ <bound>] | [ <bound> : <bound> ]
   <bound> ::= <identifier> | <INT-denotation>.
```

All parameters of mode ROWS or REF ROWS must appear in a array-descriptor with the appropriate number of bound pairs.

The <identifier> in a <bound> must identify a parameter of the ALGOL 68 routine.

2.5.4.5 FORTRAN interface example.

The FORTRAN interface may be illustrated by the following example:

```
# ALGOL 68 text#
   INT n,m;

   PROC pa = ([,]REAL rr, INT k,m) REAL:
     PR descriptor rr [1:k,1:m] PR
     rr[1,1] * r[k,m] - rr[k,1] * rr[1,m];

   # the procedure pa is equivalent to the FORTRAN routine:

      REAL FUNCTION pa(rr,k,m)
      DIMENSION  rr  (k,m)
      pa = rr(1,1) * rr(k,m) - rr(k,1) * rr(1,m)
      END
   #
   PROC f = (INT i, PROC([,]REAL rr, INT l,m)REAL p) STRING :
     PR FORTRAN ff7 12 PR SKIP;
   STRING st := f(n, pa);
# end of ALGOL 68 text #

C FORTRAN text
      CHARACTER*12 FUNCTION ff7(i, pa)
      EXTERNAL pa
      COMMON /rrr/ rr(7,9)
      x = pa(rr, 7 ,9)
      IF x < 1.0 THEN
        ff7 = 'SMALLER '
      ELSE
        ff7 = 'NOT SMALLER'
      IFEND
      END
C END OF EXAMPLE
```

The procedure "f" is declared to be a procedure written in FORTRAN with entry point name "ff7" and returning a string of 12 characters.

3 COMPILER DESCRIPTION.

In this chapter an overview of the compiler is presented.

From the rich literature on compiler construction we mention only "Compiler construction" by Waite and Goos [39], "Compilers: principles, techniques and tools" by Aho, Sethi and Ullman [40] and "Compiler construction for digital computers" by Gries [41].

Early in the project it was decided to adapt the code generator of an existing compiler. We had to chose between the FORTRAN and the SYMPL code generators. The selection was influenced by the need for major changes in the code generator selected. SYMPL was preferred for a number of reasons:

- It was written in SYMPL and the FORTRAN code generator in assembler.
- It seemed less language dependent than the FORTRAN code generator.
- Expertise was available: J. Anderson of Computer Sciences Corporation, who had been involved in the development of the SYMPL code generator was available for the project, while no expertise with the FORTRAN code generator was available.

The language dependencies in the SYMPL code generator proved to be well isolated and were removed without undue complications.

As the SYMPL code generator depended heavily on the symbol table layout of the SYMPL compiler we furthermore decided to adapt the SYMPL symbol table for the ALGOL 68 compiler.

3.1 Implementation Languages.

The compiler is coded almost entirely in SYMPL, an ALGOL 58 derivative. SYMPL was selected, because it was the only implementation language available at the time and the code generator of the SYMPL compiler was written in SYMPL, so that interface problems between code generator and other parts of the compiler could be avoided.

SYMPL [48] does not support recursive procedures and functions. It supports symbolic referencing of partial words and has enumerated types in the PASCAL sense and a macro facility.

ALGOL 68 itself is used for the implementation of transput with the exception of the primitive functions.

COMPASS [49], the assembly language for the Control Data 6000 and 7000 computers is used for the parser (generated by the parser generator), for compiler input-output routines and some frequently used symbol table handling routines. COMPASS was also used to implement the remainder of the run-time system.

3.2 Design Considerations.

As the language to be implemented was not fully defined when the project was started the compiler design had to be sufficiently general and flexible to allow for changes to be incorporated into the product under construction. The resulting structure of the compiler reflects this requirement.

As an example we decided to implement a separate lexical scan over the source text preceding the mode independent parse, as suggested by B. Mailloux in his thesis [21], because we could not predict, whether a single scan for lexical and syntactic analysis, advocated by others, would still be feasible with the final language.

This first scan determines the preliminary ranges or *corrals* in the program. The set of corrals found includes all ranges that contain mode- or operation-declarations. Corrals are defined in 3.6.2.3. The problem here is, that mode and operator indicants are lexically indistinguishable and some constructs in the language cannot be parsed without knowledge of the context e.g. the sequence: "INDICANT identifier" cannot be parsed unless it is known, whether "INDICANT" designates a mode or an operator; in the former case, the construct is an expression, in the latter case a declaration.

Another example is the allocation of local generators. Branquart [20] examined in great detail the possibility of allocation of local generators on the run-time stack. At the time we had to come to a decision, it could not be ascertained whether Branquart's reasoning would be valid for the final language. We therefore decided to generally allocate local generators on the heap. Indeed some later language changes affected the scope of local generators.

3.3 Compiler global design.

We opted for a conventional multi-pass compiler structure. As mentioned before the first two passes were dedicated to the lexical and syntactic scan of the source text.

The existing SYMPL compiler contained a pass performing register assignment and code optimization and a pass to edit the generated code into the format dictated by the system loader. These two passes were to be adapted to ALGOL 68.

The remaining tasks comprise the semantic processing and code selection. Semantic processing includes mode checking and determination of coercions; code selection includes the generation of code performing the coercions.

Combining semantic processing and code selection into a single pass causes some problems in the code selection for coercions,

because in some cases, e.g. as a consequence of balancing, the coercions cannot be determined immediately after the unit has been processed. e.g. in the text:

```
(REAL x,y ; ...
IF y > 1.0 THEN 1 ELSE x FI +=: y; ...)
```

the widening coercion on "1" is recognized only when the a posteriori mode of "IF y > 1.0 THEN 1 ELSE x FI" is determined, which mode cannot determined until the operator "+=:" has been identified.
Also for some coercions generating good code is much easier if the coercion is known before the unit is elaborated. This is certainly true for the proceduring coercion, of which we did not know that it would not be included in the language.

We therefore decided to implement semantic processing and code selection in two separate passes. The technique used to make coercions on a given unit available at the beginning of code selection for that unit will be described below with the structure of the intermediate files.

The compiler thus comprises six passes of which the first three perform the lexical, syntactic and semantic analysis of the program text and the remaining three select, generate and edit the object code.

The lexical, syntactic and semantic passes are largely machine independent, the major exception being the conversion of constants in the source to their internal representation.

The code selection phase depends on the instruction set of the Cyber machine, the code generation on the Central Processor and the Editor on the system loader.

3.4 Compiler Structure.

In the CDC 6000 series, memory is a scarce commodity. The memory available to a program is limited to 2^{17} or 131072 words. Especially for interactive users of the compiler its memory requirement had to be modest. As a result of negotiations in the Steering Group a memory requirement of 25000 words for the compilation of a program of a few hundred lines was considered acceptable.

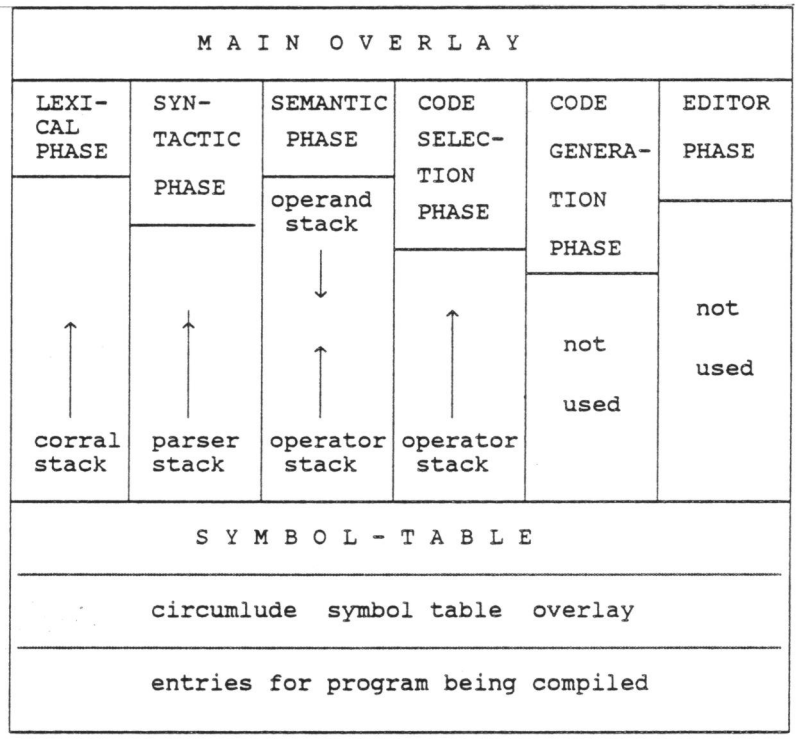
We decided to organize the compiler in overlays. The overlay technique was well proven, since all existing compilers for the CDC 6000 series used overlays to restrict memory requirements. Dynamic loading of program modules on the other hand was not supported by the operating system at the time.

The compiler consists of a main overlay , seven primary and four secondary overlays. The first six primary overlays each perform a

pass over the program in some intermediate form; the seventh primary overlay contains routines to produce edited dumps of the symbol table and of the intermediate files for debugging purposes. The four secondary overlays contain the symbol table entries resulting from the compilation of the system circumludes.

Communication between the overlays is effected by means of five intermediate files and a single table, called symbol table.

The allocation of the overlays and of the common symbol table, as well as the stacks used by some of the passes is shown in the following diagram:



3.5 Symbol Table.

The symbol table is allocated beyond the largest overlay and remains allocated for the duration of the compilation.

At the beginning of compilation, the symbol table is initialized by loading the symbol table overlay corresponding to the circumlude selected; a system circumlude is loaded from the same

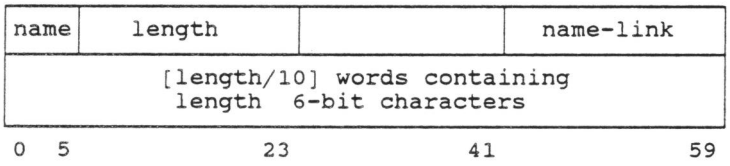
residence as the compiler, a user circumlude from the file specified by the user.

The symbol table contains entries of the following types:

type	ALGOL 68 entity	Object code entity
NAME	identifier indicant denotation	constant
CORR RNGH	corral range	range-header
PRIO OPER MODE	priority-declaration operation-definition mode-declaration	procedure-descriptor procedure-descriptor
IDEN	identity-declaration variable-declaration	addressable value addressable variable
MRW FIELD UMEM	mode-representation-word field union constituent mode	offset in struct
CONS ADCN XCON RELC	denotation	constant address constant relocatable constant relocator for xcon
ERROR	error	
LABL	label	label in code

All entries in the Symbol Table occupy an integral number of 60-bit words and contain a 6-bit field called "CLASS", that contains a code defining the type of the entry.

NAME-entries are the only variable length entries in the table and consist of a one word header and a variable length trailer as follows:



Name occupies the CLASS-field and defines the type of the entry.

Length is an integer value denoting the number of characters of the entity represented by the NAME-entry.

Name-link is an index to the symbol table entry currently identified by the indicator.

A simple 7-bit hash algorithm is employed to maintain the name table which is embedded in the symbol table. This algorithm was inherited from the SYMPL compiler and recoded in COMPASS to enhance performance. An overview and evaluation of name list algorithms is given by Grune [28].

Entries of most other types occupy two words according to the following layout:

type	next-entry	sequence-link	name-link
bits	next-outer	mode-link	location

0 5 23 41 59

The use of all fields but the CLASS-field may vary from type to type and even from phase to phase. The symbolic names in capital letters and the most frequent usage of the fields are tabulated below.

type CLASS	defines the type of the entry.
next-entry NEXT	index of the symbol-table entry representing the next entity (e.g. mode or operation declaration) in the same range in the program.
sequence-link ASEQ	type-dependent.
name-link NLNK	address of the name-entry associated with the entry.
bits 0-5 BIT0-BIT5	used as six separate one-bit fields.
next-outer NXTO	address of the symbol-table entry of the entity with the same name that becomes visible, when the scope of this entry is exited.
mode-link MDLK	address of the symbol-table entry representing the mode of the entity represented by this entry.
location LOCN	the address in the object program of the entity represented by this entry: for an iden-entry this would be the relative address of the variable in the stack-segment of the procedure containing its declaration; for a field-entry this would be the offset of the field in the structure containing the field

Entries retained from the SYMPL compiler, such as ADCN- and CONS-entries show a somewhat different layout.

3.6 Compiler Overlays.

The main functions of the main overlay and the first six primary overlays are listed below.

Main overlay:

- overall control,
- overlay loading,
- symbol table handling.

Lexical phase:

- Command statement interpretation,
- Initialization of the symbol table,
- Reading source code and writing source listing,

. Token and pragmat recognition,
. Preliminary range or corral recognition,
. Encoding of ICF-macros.

Syntactic phase:

. Mode independent parsing,
. Creation of symbol table entry from declarations,
. Building mode trees for all declared modes.

Semantic phase:

. Mode equivalencing,
. Mode checking and coercion determination,
. Writing of circumlude symbol table.

Code generation:

. Mode dependent interface routine generation,
. Memory allocation in stack segments,
. Expansion of ICF-macros,
. Generation of registerless code.

Code optimization and register assignment:

. Building of a dependency tree,
. Prioritizing the instructions,
. Register allocation,
. Memory allocation.

Editor:

. Generation of loadable module,
. Error listing generation,
. Object code listing generation.

3.6.1 Main overlay.

In the main overlay the loading and execution of the compiler overlays is controlled. The main overlay also contains some common data and routines.

3.6.2 Lexical phase.

In the first overlay the ALGOL 68 compilation-control statement is decoded, the symbol table is initialized and a lexical scan over the source text is performed. The compilation-control-statement parameters are described in Appendix B.

Unless the standard circumlude is being compiled, the overlay containing the symbol table resulting from the compilation of the circumlude specified on the compilation control statement is

loaded. If however the standard circumlude is being compiled the symbol table is initialized to contain entries for fundamental entities such as the modes INT, REAL, CHAR, BITS and BYTES.

In the lexical scan the following functions are performed:

1. Input of the source text and output of the source listing.

2. Recognition of basic tokens and removal of comment and blanks outside of STRING-denotations.
   Replacement of simple syntactic units, such as identifiers, field-names, labels, indicants and denotations by an index to the appropriate entries in the symbol-table.

3. Analysis of the parenthesis structure, determination of preliminary ranges or corrals and recognition of formal-parameter-plans.

4. Recognition of prio-, operation- and mode-declarations.

5. Processing of pragmats.

6. Output of the transformed source text as IL1 (Intermediate Language 1).

3.6.2.1 Input and printout of the source text.

A single subroutine prints the current source line and reads the next source line. Normally the characters in the source file are in the 6-bit CDC character set and must be converted into a 12-bit superset of ASCII. General images are decoded. A character set table is included in appendix A.

3.6.2.2 Treatment of simple syntactic units.

For identifiers, field names, labels and indicants a NAME-entry is generated using a 7-bit hash algorithm. For INTREAL- and STRING-denotations a CONS-entry is generated. A CONS-entry representing a denotation refers to a NAME-entry for the corresponding internal representation and to a MRW- or mode-representation-word-entry for the mode of the denotation.

FORMAT-denotations are converted according to the method of Goos [18] into a string and a row display; the string contains the format-patterns and the elements of the row display represent the dynamic parts of the format; the mode of these elements is:

```
UNION (PROC INT, PROC [ ] CHAR)
```

3.6.2.3 Analysis of the parenthesis structure.

Branquart [19] introduced the concept of preliminary range or corral. A corral is defined as a lexical construct that may yield a range containing a mode-, prio- or operator-declaration In the following constructs one or more corrals are found. Corrals are indicated by "--" in the table.

bold tag form	brief form
BEGIN -- END	(--)
IF -- THEN -- ELSE -- FI CASE -- IN OUT -- ESAC	(-- | -- | --)
IF--THEN--ELIF--THEN--ELSE--FI CASE--IN OUSE--IN OUT--ESAC	(-- | -- |: -- | -- |--)

These are all constructs, that may yield a range containing mode-, operation- or prio-declarations. For each corral identified, a CORR-entry is generated, the index of which is output to the IL1-file immediately following the opening symbol of the corral.

3.6.2.4 Recognition of formal-parameter-plans.

In ALGOL 68 formal-parameter-plans cannot be expressed in a SLR(k) grammar for any finite k.

As early as 1973 Moudry e.a. suggested to modify the language in order to remedy the situation, but their suggestion was not incorporated into the language.

In the first version of the compiler formal-parameter-plans were recognized in a semantic routine in the syntactic phase using a indefinite look ahead.

The author later designed and implemented the recognition of formal-parameter-plans in the lexical scan.

A formal-parameter-plan has the structure:

```
( -- ) virtual-declarer ;
```

If a construct contains a proc-, struct-, union- or ref-symbol it is easily recognized as a virtual-declarer. Otherwise it consists of an indicant possibly preceded by one or more rowers, in which case the construct is recognized as a formal-parameter-plan by

the colon following the indicant.

Some possible forms of formal-parameter-plans are given below, where the caret in each form shows at which point the construct is recognized as a formal-parameter-plan.

```
( -- ) [  ] [  ]...[  ]  PROC ( -- ) <indicant> :
                         ^

( -- ) [  ]  STRUCT ( -- ) :
             ^

( -- )  UNION ( -- ) :
        ^

( -- )  REF ...      :
        ^

( -- ) [ ] [ ] ... [ ] <indicant> :
                                  ^
```

If a formal-parameter-plan is detected, the formal-flag is set in the CORR-entry of the corral corresponding to the first "(--)" construct in the examples above.

3.6.2.5 Recognition of prio-, operation- and mode-declarations.

For each of these declarations a PRIO-, OPER-, or MODE-entry is generated and chained from the CORR-entry representing the current corral.

3.6.2.6 Processing of pragmats.

All pragmats supported and their formats are summarized in the table below. The formats containing "..." are not necessarily complete as shown.

Category	format	function
Source listing control	PR list PR PR nolist PR PR page PR PR eject PR PR state PR PR nostate PR	turns on listing. turns off listing. page eject on listing. page eject on listing. status info on listing. no status info
Switching stropping regime	PR point PR PR upper PR PR res PR PR stropped PR PR flagged PR	point-stropping. upper-stropping. res-stropping. stropped-stropping. flagged-stropping.
Source marking	PR stop PR PR prog PR	end of compilation unit. particular program.
Separate compila-tion control	PR xdef ... PR PR fedx PR PR xref ... PR PR fortran ... PR PR descriptor PR	start of external routine-text. end of external routine-text. external reference. FORTRAN routine. bounds of multiples passed from FORTRAN.
Operator defini-tion	PR inline ICF-macro PR PR same ... PR	new ICF-macro for operator. existing ICF-macro.

The listing control and stropping control pragmats affect the processing in the lexical scan itself; their processing results in the setting of switches only.

The source marking pragmats are basically substitutes for syntactic tokens not provided in the ALGOL 68 language. The stop- and prog-pragmat are translated into specific IL1 items, which are terminal symbols in the grammar for the parser.

The first four separate compilation pragmats also show a simple format:

```
PR xdef {<external-name>} PR
PR fedx  PR
PR xref <external-name> PR
PR fortran {<external-name>} {<length>}  PR
```

<external-name> is a name acceptable to the Cyber loader and
<length> is an INT-denotation, used in the declaration of a FORTRAN CHARACTER function to define the length of the result.

Each of these pragmats is translated into a specific item into

the IL1 output file with the external-name as parameter. These items are terminal symbols for the parser.

The descriptor pragmat is used in the declaration of routines, that can be passed as parameter to a routine written in FORTRAN. The purpose of the descriptor pragmat is to define the bounds of parameter arrays. Here it may suffice to state that it takes the place of a FORTRAN dimension statement applied to array parameters.
The format of this pragmat is:

```
    PR descriptor <array-descriptor> {,<array-descriptor>} PR
```

where

```
<array-descriptor> ::= <identifier> <bound-pair> {,<bound-pair>}.
<bound-pair>       ::= [<bound> { : <bound> } ].
<bound>            ::= <identifier> | <INT-denotation>.
```

An example of a descriptor-pragmat:

```
      PR descriptor ar1 [1 : j], ar2 [7 : 13] ,
                    ar3 [j , i : k, m : 17] PR
```

Only the beginning and end of a descriptor-pragmat are translated into specific IL1-items. The array-descriptors are processed as if they were not contained in a pragmat.

The inline-pragmat is used to define an ALGOL 68 operator in terms of ICF-code and takes the form:

```
    PR inline
      ICF-macro-header
       <ICF-instruction>
      {<ICF-instruction>
                          }   PR
ICF-macro-header ::= head <length> <result> [S] {<macro-number>}.
<length>         ::= INT-denotation. # number of instructions#
<result>         ::= INT-denotation. # number of result words#
<macro-number> ::= INT-denotation. # identifies the macro  #
```

If S is specified multiple operators are defined, as described in Appendix D.

```
ICF-instruction ::=
      {<res-nr>} {<label>} <ICF-code> {<operand> {,<operand>}}
```

<res-nr> is a INT-denotation, that specifies which word of the result is defined by this instruction,
<label> is a single letter, that may be used to refer to this instruction from within the macro and
<operand> specifies the first or second operand of the instruction.

Operands may specify one of the following:

operand type	format	comment
argument	/<int>	<int> is the argument number
instruction of the macro	*-n <label>	n-th preceding instruction instruction labeled "label"
external entity	$<ext-name>	
compile time variable	r<d>, where <d>::= 1|2|3|4.	r1 the current THEN-label r2 the current ELSE-label r3 the stack offset r4 the current line number
direct constant	=<denotation>=	

The pragmat is translated into an IL1 item, specifying the index of a new NAME-entry containing one word for each ICF-instruction.

The same-pragmat is used in the same manner as the inline-pragmat and has the format :

PR same <macro-number> {<spec>} PR

<macro-number> specifies the ICF-macro, in the header of which the same <macro-number> is specified.

<spec> is a single letter "A" or "T".

If <spec> is omitted the same-pragmat specifies the ICF-macro identified by <macro-number>.
In this case the same-pragmat provides a shorter notation for the definition of an operator, for which the same code is to be generated as for some predefined operator.
A same-pragmat without <spec> is typically used for the definition of a number of operators, that are defined in the standard prelude by one and the same operation-definition, using the extended syntax described in RA68 10.1.3 step 3.
As an other example in order to define the operator "=" for INT operands to be translated into the same code as "=" for REAL operands and assuming that the latter has been defined by means of an ICF-macro with <macro-number> = "56", we use the definition:

```
OP (INT, INT) BOOL = = PR same 56 PR;
```

Same-pragmats with <spec> = "A" or <spec> = "T" are intended to be used in the definitions of so-called "and becomes" and "to" operators.

For the operator "ABOP" defined by:

```
OP (AMODE, BMODE) AMODE ABOP =
PR inline ICF-macro-ab PR SKIP;
```

the associated "and becomes" operator "ABOPAB" is defined by:

```
OP ABOPAB =
(REF AMODE a, BMODE b) REF AMODE : a := a ABOP b.
```

Similarly for the operator "BAOP" defined by:

```
OP (BMODE, AMODE) AMODE BAOP =
PR inline ICF-macro-ba PR SKIP; .
```

the associated "to" operator "BAOPTO" id defined by:

```
OP BAOPTO =
(BMODE a, REF AMODE b) REF AMODE : b :=  a  BAOP b;.
```

The same-macro with <spec> = "A" or <spec> = "T" allows to define inline code for the "and becomes" or "to" operators in terms of the ICF-macro associated with the base operator.

Specifically, if ICF-macro-ab contains a <macro-number> say "abop-num", we may define "ABOPAB" by:

```
OP ABOPAB = (REF AMODE a, BMODE b) REF AMODE :
PR same abop-num a PR SKIP;
```

and if ICF-macro-ba contains a <macro-number> say "baop-num", we may define "BAOPTO" by:

```
OP BAOPTO = (BMODE a, REF AMODE b) REF AMODE :
PR same baop-num t PR SKIP;
```

The same-macro is used frequently in the implementation of the standard circumlude, saving a few hundred lines of code.

The same-pragmat results in an IL1 item specifying the NAME-entry corresponding to the ICF-macro identified by "macro-number" together with an indication of the <spec> value specified.

3.6.2.7 Output to the IL1-file.

The IL1-file contains 20-bit entries, composed of a 3-bit type and 17-bit value field.

bit positions				
0 2	3 9	10 19		
type	value field		entity represented	example
0	0	basic symbol	basic symbol	BEGIN
1	line number		line number	112
2	CORR-entry index		corral	(
3	NAME-entry index		identifier	maxint
4	NAME-entry index		indicant	PLUSAB
5	CONS-entry index		denotation	3.8
6	NAME-entry index		MONAD symbol	/
7	NAME-entry index		NOMAD symbol	++:=

3.6.3 Syntactic phase.

In the second scan a mode independent parse of the source program is performed. The priority dependent parse of formulas is deferred to the third scan in order to simplify the grammar.

The parser was constructed by means of a parser generator, of which the input consists of a SLR(3) grammar with interspersed semantics and the output is a 10000 line parser program in the assembler language COMPASS.

The parser generator is based on a design of De Remer [15]. The parser is described in detail by Van Oostrum [23]. A similar parser generator generating code rather than tables is described by Penello in "very fast LR Compiling" [36].

Later the author improved the parser generator to accept interspersed error numbers in the grammar in order to obviate renumbering of error messages for every change in the grammar.

The syntactic scan parses the program on intermediate file 1 (IL1) that was produced by the lexical scan and executes in each reduce state the indicated semantic action and calls the recovery

routine with the internal state-number as parameter, whenever a syntax error is detected.

The following sections provide more details on:

. The input routine.
. Formulation of the grammar.
. The parser program.
. Declarations and ranges.
. Modes.
. Parallel clauses.
. Output of the IL2-file.

3.6.3.1 Input routine.

The IL1-file is read and the information read is converted to the appropriate parser-input format. Every call of the input routine supplies a single symbol, together with up to two parameters, which the parser can pass to the appropriate semantic routine.

The different types of IL1-entries, listed above are processed as follows:

0. ALGOL 68 symbols are passed unchanged.

1. Line numbers are processed outside of the parser.

2. Corrals are not passed to the parser; rather the corral is opened, i.e. the associated declarations are made accessible; corrals are closed by means of the semantic actions specified in the grammar. If the formal-flag in the CORR-entry for the corral is set, a formal-symbol is inserted in the parser input.

3. Identifiers yield a "TAG" symbol with the NAME-entry as parameter.

4. Constants yield a "CONS" symbol with the CONS-entry as parameter.

5. Indicants are identified with accessible declarations (contained in an opened corral) and yield either a mode-indicant-symbol with the MODE-entry as parameter or an operator-indicant-symbol with the NAME-entry as parameter.

6. Monad yields a monad-symbol with the NAME-entry as parameter.

7. Nomad yields a nomad-symbol with the NAME-entry as parameter.

3.6.3.2 Formulation of the grammar.

In the Revised Report ALGOL 68 is defined using a two level grammar. From this grammar a BNF grammar of a superlanguage is derived by replacing production rules that produce an unlimited set of rules by a single production rule. In this process all mode dependent restrictions are lost. These restrictions are enforced by the semantic pass of the compiler. The grammar is listed in appendix C. Some specific issues are:

The production rule for a formal-parameter-plan begins with a formal-symbol that does not appear in the IL1 file, but is inserted into the parser input by the input routine.

As each indicant is identified with a declaration (within a corral) its type (operator or mode) is known and constructs like "TAG x" can be parsed unambiguously.

The priority of operators is ignored at this stage reducing the number of production rules.

Labels in serial- and enquiry-clauses are syntactically allowed, but semantically detected.

3.6.3.3 The parser program.

Parsing proceeds bottom up. The input is read until a production rule is recognized. The rule is then replaced by an indication of the notion represented by the rule, after a call of a semantic routine is made.

The parser produces a representation of the production tree of the program in reversed Polish notation, called IL2 (Intermediate Language 2). In IL2 a notion is represented by an operator, its direct branches by the associated operands. Operands are operators or terminal symbols. Terminal symbols are numeric values or indices in the symbol table, referring to entries representing e.g. ranges, modes, identifiers,or indicants.

Notions, associated with lists or sequences of operands are represented by operators with a variable number of operands. An example is the COLL(n)-operator denoting a collateral-clause with n units.

Some operators denote a common part of different notions. The RANGE-operator that occurs where a range is formed, e.g. in a choice-clause or a while-clause, is an example.

In a few cases a single operator may represent different notions. The CLOSED-operator for example represents a closed clause or a collateral clause. The further analysis required in such cases is performed in the semantic scan.

3.6.3.4 Declarations and ranges.

For each declaration an entry in the symbol table is generated.

declaration	entry generated
identity-definition variable-declaration parameter specification specification in conformity-clause specification in for-part	IDEN-entry
operation-definition	OPER-entry
mode-declaration	MODE-entry

If a circumlude contains a prio-declaration, but no operation-declaration for the operator concerned, a (dummy) OPER-entry is generated for that operator. In all other cases the priority is stored in the already existing corresponding OPER-entry.

An RNGH-entry, representing a range in the source program, is generated upon detection of the first declaration in the current corral.

The generated IDEN, OPER an MODE entries generated are linked to the RNGH-entry of the current range.

The corral and range stuctures produced are largely but not completely equal. A corral, but no range is produced for the open-symbol in : "(a+b)". A range, but no corral is produced for "FOR i TO". The redundancy implied facilitates error recovery.

3.6.3.5 Modes.

The syntactic scan builds for every declarer a tree structure of MRW (mode representation word)-entries in the symbol table. Declarers of mode PROC, UNION and STRUCT give rise to additional RNGH- and IDEN-, FIELD- or UMEM-entries.

A MRW-entry contains a.o. PIVA-, AUXL-, and MDLK-fields with contents as tabled below:

Declarator	PIVA	AUXL	MDLK
INT	INT	(1)	0
REAL	REAL	(1)	0
BITS	BITS	(1)	0
BYTES	BYTES	(1)	0
VOID	VOID	0	0
BOOL	BOOL	0	0
CHAR	CHAR	0	0
union of rows	ROWALL (2)	0	0
input	INTYPE (3)	0	0
output	OUTTYPE (3)	0	0
error	ERROR (4)	0	0
REF AMODE	REF	0	MRW-entry of AMODE
[ ] AMODE	ROW	number of dimensions	MRW-entry of AMODE
PROC	PROC	number of parameters	RNGH-entry of parameter-range
UNION	UNION	number of constituents	UMEM-entry of first constituent
STRUCT	STRUCT	number of fields	FIELD-entry of first field

(1) number of LONG-symbols.
(2) union of all ROW-modes, the operand mode of LWB and UPB.
(3) the so-called primitive transput-modes.
(4) used for syntactically incorrect declarers to avoid propagation of errors.

UMEM- and FIELD-entries are chained.

The use of the REFM-, ROWM- and ROWR-fields in MRW-entries is described below in section 3.6.4 on the semantic phase.

3.6.3.6 Parallel clauses.

A parallel clause has the form:

```
PAR (unit-1, unit-2,..., unit-n)
```

where unit-k (k=1,2,... n) is a unit to be executed in parallel with the other units.

The IL2 output corresponds to the source code :

```
Execpar ((PROC VOID: unit-1, 0), (PROC VOID: unit-2, 0),...
        (PROC VOID: unit-n, 0) )
```

where execpar is a predefined routine of mode:

```
PROC ([ ] STRUCT (PROC VOID x, INT y)) VOID
```

The INT y field is used in execpar only. Execpar is discussed in sections 4.5.2.3 and 4.5.3.4.

3.6.3.7 Output of the IL2-file.

The IL2 file contains a representation of a parse tree of the program in reversed Polish notation. Intermixed with the representation are some special entries e.g. line numbers.
The tree representation consists of two kinds of entities: operators and operands.

1. Operators are taken from a fixed set "QIL2OP". An operator is always accompanied by its operand count. An operator entity is denoted by its name and followed by its number of operands enclosed in parentheses, e.g. COLL(4), where COLL designates the collateral clause operator. COLL(4) might represent a structure display of 4 elements e.g. "(3,i+4,9,j*7)".
   Some operators always have the same number of operands e.g. RANGE (2), representing a range in the program; the operands of the RANGE operator are the unit of the range and the MRW-entry of the yield of the range. The operands of an operator are operators or basic operands.

2. Basic operands are the leaves of the parser tree. Examples are a symbol table pointer or a number (e.g. priority of an operator).

Physically the IL2 file is a stream of 20-bit bytes. The two most significant bits contain the type, i.e. one of: long-operator, short-operator, direct-operand and line-number.

An operator with at least 2048 operands is represented by two bytes, of which the first contains a 2-bit type-code=0, a 7-bit operator field and an 11-bit unused field and the second byte contains the number of operands.

An operator with less than 2048 operands is represented by a single byte containing a 2-bit type-code=1, a 7-bit operator field and an 11-bit field for the number of operands.

A direct operand requires a single byte with a 2-bit type-code=2 and an 18-bit value.

A line number uses a single byte with type-code=3 and an 18-bit line number.

The layout of IL2-entries of different types is tabulated below:

IL2	byte		representing
0	QIL2OP	0	operator with > 2047 operands
0	operand count		operand count
1	QIL2OP	operand count	operator with < 2048 operands
2	value		direct operand
3	line-number		line number

0 2 9 19

3.6.4 Semantic phase.

The third overlay contains both the mode-equivalencing routine and the semantic scan of the program text. Mode equivalencing must be performed between the syntactic and the semantic scan; We had the choice of allocating it either in the second or the third overlay or in an overlay of its own. It is included in the third overlay because this addition did not cause this overlay to become the biggest overlay. The mode-equivalencing and coercion routines are described by Van Hedel [24].

3.6.4.1 Mode equivalencing.

In "On Infinite Modes" [16] Koster describes an algorithm to determine the equivalence of any two mode declarers by induction, formulated as follows:

> "Two mode-indications specify one same mode if, under the assumption that they do, their defining declarers specify one same mode.
> A mode-indication and a declarer specify the one same mode if, under the assumption that they do, the declarer specifies the same mode as the defining declarer of the mode indication.".

The mode-equivalencing routine operates on symbol table entries only. The algorithm implemented is based on the work described in the thesis of Mary Zosel [17].

In this algorithm modes are considered equivalent, unless they are proved to be different. The modes under consideration are divided into classes. In a number of refinement steps these classes are split into classes until each class contains equivalent modes only. This process is executed in four steps:

1. All modes are placed in a class corresponding to the value of the PIVA-field of the associated MRW-entry.

2. The classes corresponding to the PIVA-values INT, REAL, BITS, BYTES, ROW, PROC and STRUCT are split based on the value of the AUXL-field of the MRW-entry.

3. The classes for PIVA = STRUCT are split according to the field names.

4. All classes are numbered. Modes in the same class receive the same mode-number.
   All classes with PIVA = REF, ROW, PROC, STRUCT or UNION are examined and split according to the rules given below.

   Classes with PIVA = REF are split according to the the mode-number of the object mode.

   Classes with PIVA = ROW are split according to the mode-number of the mode of the component of the multiple.

   Classes with PIVA = PROC are split on the basis of the mode-numbers of the MRW-entries of the parameters and the result of the procedure.

   Classes with PIVA = STRUCT are split on the basis of the mode-numbers of the modes of the fields of the structure.

   Classes with PIVA = UNION are preprocessed by reordering the constituent modes of each mode in the class in the order of their mode-number and then split on the basis of the mode-numbers of the constituent modes of the union.

Step 4 is repeated until no more classes are created.

Goos [18] correctly states (see top of page 187) that the attribution of an ordinal number with a fixed maximum length to every mode is possible only if we choose the numbers dependent on the particular program at hand. Note that the ordinals mentioned above are called mode-numbers in this implementation.

He concludes that for each variable of united mode we need to store both value and (some representation of the actual) mode and proposes to represent the mode by the ordinal attributed to the mode during compilation of the particular program.
His final conclusion, that the binding of more than one program will cause problems necessitating the use of a special-purpose linkage-editor does not hold for this implementation.

In the algorithm described above the ordering of modes within any set of modes depends on that set of modes only. If the same set is found in different compilations the same ordering is obtained in each compilation, irrespective of any other modes occurring in any of such compilations.

The algorithm therefore defines a canonical ordering of the constituent modes of any united mode. Accordingly in the internal

representation of values of mode UNION the field identifying the actual mode will contain the ordinal attributed to the actual mode by this canonical ordering rather than its mode-number.

In this implementation therefore the use of UNION modes across separately compiled program units causes no problems, since in all units the internal representation of the UNION will be the same.

The mode-equivalencing routine all modes explicitly declared in the program. Other modes may be implicitly defined as the result of row displays, slicing, coercions and selections.

In order to allow the semantic scan to equivalence such implicitly defined modes with explicitly declared modes up to three pointers are stored in entries representing modes.

Designating the mode of n-dimensional multiple of elements of mode AMODE by "ROW-n of AMODE" we store:

1. in the REFM-field of the entry for mode AMODE a pointer to the entry for REF AMODE.

2. in the ROWM-field of the entry for mode AMODE a pointer to the entry for ROW-i of AMODE, where i is the lowest value of n, for which the mode ROW-n of AMODE is represented in the mode table.

3. in the ROWR-field of the entry for each mode ROW-j of AMODE, a pointer to the entry for ROW-i of AMODE, where i is the lowest value of n greater than j, for which the mode ROW-n of AMODE is represented in the mode table.

3.6.4.2 Semantic scan.

The following subsections describe:

. IL2 input driving the scan.
. Range open and close ; identification.
. Stacks for IL2 operators and operands.
. Mode checking and coercion determination.
. Optimized variables.
. IL3 output.

3.6.4.2.1 IL2 input driving the scan.

The semantic scan processes the IL2 file backwards and thus effectively reads the program backwards in prefix Polish notation.

Processing of the items on the IL2 file is coordinated by IL2-operator. For each IL2-operator three separate routines exist. These routines are activated at the input of the operator,

after processing each non-last operand and after processing the last operand, respectively. This scheme was adopted in preference to a single routine, because the implementation language used does not support recursive procedures.

3.6.4.2.2 Range open and close ; identification.

At range entry all IDEN-, OPER- and MODE-entries representing declarations in that range, are chained into a list the head of which is referred to by a field in the NAME-entry for the identifier, operator-indicant or mode-indicant. Multiple OPER-entries so introduced in one list (i.e. entries representing multiple operation-declarations for the same indicant in the range just opened), are checked in order to detect related operators.

At any time the first entry in each list found via a NAME-entry thus represents a currently valid declaration. For any identifier or mode indicant there is only one declaration in force. For operators the entries in the list of OPER-entries are searched for an operator with appropriate operand-modes. Because of the check for related operators at range entry at most one such entry will be found.

3.6.4.2.3 Stacks for IL2 operators and operands.

During this scan two stacks are maintained: the operator-stack and the operand-stack. The operator-stack contains an entry for each IL2 operator being processed. The entry contains a.o. :

IL2-operator,
number of operands of the IL2-operator,
number of operands in the operand stack,
operand counter.

The operand-stack consists of fixed size entries, one for each operand. An entry contains a.o.:

. Type of the operand, which may be : collateral, balance, denotation, identifier, nil, skip, formula, etc.,
. Number of suboperands, for collateral or balance type operands,
. Index in the symbol table of the identifier or denotation,
. Mode of the (sub)operand,
. IL3-index of the operand.

For collateral or balance type operands the entries for the suboperands are stacked after the main entry; since a suboperand may again be a collateral or balance type operand the structure of the stack entry is recursively defined.

3.6.4.2.5 Maintaining the mode table.

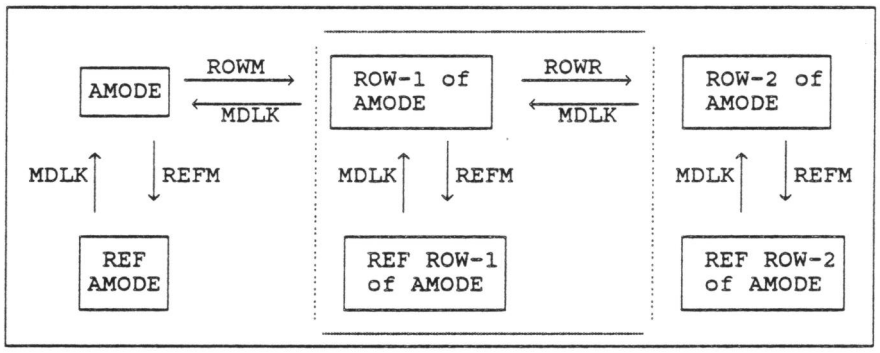
The *a priori mode* of an operand in most cases is a mode represented in the mode table. A few constructs however may have a not explicitly declared mode. Such constructs actually definelare a mode and we must either find the entry for the equivalent mode in the mode table or generate a new entry for it. We will discuss the slicing construct as an example.

The declarations:

```
MODE AMODE = ....;
[7,11] AMODE r2_of_amode;
```

give rise to MRW-entries for the modes: "AMODE", "REF AMODE", "ROW-2 of AMODE" and "REF ROW-2 OF AMODE".
An MRW-entry for the mode "ROW-1 of AMODE" may or may not exist at the time we encounter the slice: "r2_of_amode[1,]".

The figure below shows the MRW-entries concerned. MRW-entries are shown as boxes. An arrow with a field name shows a pointer to the entry pointed to by the arrow in the field denoted by the field name in the entry represented by the box from where the arrow originates. The use of the MDLK-field was shown in the table in section 3.6.3.5 and that of the REFM-, ROWM- and ROWR-fields in section 3.6.3.5.



The complete figure shows the case, where the mode of the slice, which is "ROW-1 of AMODE" is represented in the mode table and the figure without the area in the dotted lines shows the situation, where this result mode is not represented in the table. In the latter case the changes in the mode table made in the processing of the slice update the table to the situation depicted by the figure including the area within the dotted lines.

3.6.4.2.5 Mode checking and coercion determination.

Mode checking proper is done in the coercion routine. This routine is implemented as a boolean function with three parameters: context (strong, firm, meek, weak or soft), index of coercend in the operand-stack and *a posteriori mode*.

The coercion routine yields true, if the operand designated by the index in the operand-stack can be be coerced to the a posteriori mode in the given context; in that case the associated coercions are placed on the operator-stack.

Coercion handling is recursive because the coercends are recursively defined. A recursion-stack is maintained on the operand-stack. Dereferencing coercions give rise to another level of recursion because the dereferencing of a multiple being a component of a multiple requires its own coercion-entry. The dereference-coercion-stack is maintained on top of the coercion-stack on the operand-stack and, of course, the resulting coercion-entries are placed on the coercion-stack on the operator-stack.

The following types of coercion-entries are distinguished:

TYPE	Coercion
DEPROC	Deproceduring
EMPTY	SKIP to any mode
ROWDIS1	Row display of non-multiple values
ROWDIS2	Row display of multiple values
ROWD1	Rowing of multiple value to multiple value of higher dimension
ROWD2	Rowing of reference to multiple to reference to multiple of higher dim.
ROW1	Rowing of non-multiple value
ROW2	Rowing of a reference to non-multiple
STRUDIS	Structure display
UNITE1	Uniting of non-union
UNITE2	Uniting of united value
UNROWALL	Union of multiple to mode ROWALL
VACUUM	Create a vacuum
WIDEN1	INT to REAL
WIDEN2	INT to COMPL
WIDEN3	REAL to COMPL
WIDEN4	BITS to [ ] BOOL
WIDEN5	BYTES to [ ] CHAR
DEREF	Dereferencing
DEREF2	Secondary entry indicating copy of multiple value

The coercion algorithm consists of the following steps:

1. Initialization of recursion-stack and coercion-stack.

2. For a balance-type operand, set up loop over suboperands, propagating the context.

3. For a collateral-type operand:

   If the a posteriori mode is VOID, generate a voiding-coercion.

   If the a posteriori mode is a STRUCT-mode, generate a STRUDIS-coercion and set up loop over suboperands with context STRONG and mode of corresponding field as a posteriori mode.

   If the a posteriori mode is a ROW mode, generate ROWDIS1-coercion and set up loop over suboperands with context STRONG and mode of the component as a posteriori mode.

   Otherwise yield false.

4. Process special operand types, such as jump, nil, skip and vacuum.

5. Process a posteriori ROWALL (another recursion for size!)

6. Mark the MRW-entries of all modes, reachable by repeated dereferencing and/or deproceduring of the a priori mode.

7. If step 6 marked the a posteriori mode, go to step 11.

8. For a STRONG context: test whether widening of one of the marked modes yields the a posteriori mode; if so, generate the appropriate WIDENx-, (x=1..5) coercion-entry and go to step 11.

9. For a STRONG context and an a posteriori mode of the form [.] AMODE or REF [ ] AMODE, generate the appropriate coercion-entry, replace the a posteriori mode by AMODE or REF AMODE, respectively and go to step 7.

10. For a FIRM or STRONG context and a united a posteriori mode test whether one of the marked modes can be united to the a posteriori mode; if so generate the appropriate coercion-entry; otherwise yield false and exit.

11. Generate the necessary dereferencing- and deproceduring-coercion-entries and clear the marks set in step 6.

The generated coercion-entries all contain the IL3-index of the operand to which the coercion is to be applied. The output of the coercions on IL3 file is not done in the coercion routine itself, because in some cases, e.g. operator identification a coercion is finally accepted only when multiple operands can be coerced. The

IL2-operator processing routine having checked the modes through a series of calls of the coercion routine invokes a coercion output routine to write the coercion entries to the IL3 file.

3.6.4.2.6 Optimized variables.

In a later version of the compiler so-called optimized variables have been implemented. Optimized variables are variables where the value rather than the reference is placed in the run-time stack.
In this implementation references point into the heap, a reference is allowed to point into the stack however when no scope checking is needed. A variable may be optimized if it is used only as the target in an assignation, as a coercend of dereferencing or as an actual parameter corresponding to an optimized formal variable.
To verify the last condition all pairs of actual and formal variables are stored in a table. After processing of the entire source program the non-optimized predicate is propagated.

3.6.4.2.7 Calling a FORTRAN routine.

Every PROC-mode appearing as the mode of a parameter in a call to a FORTRAN routine is placed in a list. This list is used in the code-generation phase for the generation of mode dependent FORTRAN interface routines as described in section 3.6.5.1.

3.6.4.2.8 IL3 output.

A representation of the program in postfix Polish notation with interspersed coercion entries is written on the IL3-file.

The fourth scan reads the IL3 file backwards as a representation of the program in prefix Polish notation. Coercion-entries are always written on the IL3-file after the operands to which they apply. In the code-generation scan therefore the coercions to be applied to an operand are available before the operand is read from the IL3-file.

The physical structure of the IL3-file is similar to that of the IL2-file, the differences being, that IL3-operators are taken from a fixed set QIL3OP and that the IL3-file contains coercion-entries.

3.6.4.3 Circumlude symbol table output.

If a circumlude is being compiled the symbol table is written to the file specified on the compilation control statement.

3.6.5 Code generation.

This section describes the fourth phase of the compilation process.

The fourth phase performs:

. FORTRAN mode dependent interface routine generation.
. Code generation scan.

3.6.5.1 Mode dependent FORTRAN interface routine generation.

Each mode PROC (...) AMODE, that occurs as the mode of a parameter in a call of a FORTRAN routine was placed in a list during the semantic scan.

For each mode in that list an internal subroutine is generated, that converts the parameter list of a FORTRAN routine of that mode into its ALGOL 68 equivalent. The use of these internal subroutines is described in sections 3.6.5.5 and 4.5.1.3.

3.6.5.2 Code-generation scan.

The following subsections describe:

. IL3 input driving the scan.
. Operator and coercion stacks.
. Memory allocation in stack segments.
. Generation of registerless code.
. Calling a FORTRAN routine.
. Expanding of ICF-macros.

3.6.5.2.1 IL3 input driving the scan.

The code-generator scan is organized by operator read from the IL3 file, that contains a representation of the program in prefix polish notation. The interspersed coercion-entries are input and chained for later access. Coercion-entries precede the operands they refer to on the IL3 file. At the time of reading an operator from the IL3 file, the associated coercion-entries are at the head of the coercion-entry-chain.

For each IL3-operator the code generator contains three separate routines. These routines are activated at the input of the operator, after processing each non-last operand and after processing the last operand, respectively. This scheme, which is similar to the one employed in the semantic scan was adopted in preference to a single routine, because the implementation language used does not support recursive procedures.

3.6.5.2.2 Operator and coercion stacks.

An operator stack is maintained containing a fixed size entry for each operator being processed. Each entry contains a.o.

- the IL3-operator,
- the number of operands in IL3,
- an IL3-operand counter,
- up to two operand descriptors.

ALGOL 68 values are divided in large and small values. Values occupying more than two words are considered large and are handled as an area in memory . Large values are loaded in the run-time stack and are represented in the operand descriptor by their relative address in the current stack frame. An ALGOL 68 value occupying one or two words is considered small, is handled as the contents of one or two registers and is represented in the operand descriptor by one or two ICF-numbers. An ICF-number is the index of a previously generated ICF-instruction on the ICF IL4 (Intermediate Code File).

An operand descriptor contains the subfields:

- symbol table index or stack offset,
- one or two ICF-numbers,
- size of the ALGOL 68 value in words,
- type of the operand (normal, in-stack, optimized, ...).

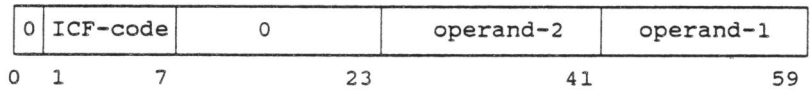
Operand descriptors reflect the current state and residence of the operand. If a subroutine call is to be generated a scan is made through the operator stack and all operands currently represented by ICF-numbers are forced on the run-time stack; this is necessary because the execution of a subroutine voids all register contents.

3.6.5.2.3 Memory allocation in stack segments.

Upon range entry the variables, modes, and operators declared in the range are allocated in the current stack segment. Intermediate results are allocated at the top of the current stack segment.

3.6.5.2.4 Generation of registerless code.

The code generator produces registerless code, also called ICF (Intermediate Code File) ICF entries contain an operator and up to two operands;



ICF operands are of two types:

1. a reference to some preceding ICF-instruction specifies the result of that instruction.

2. symbol table index or a constant value depending on the ICF operator. A symbol table index specifies the entity denoted by the symbol table entry e.g. a denotation or a variable and a constant value may have different usages, e.g. the shift count in a shift instruction.

ICF operators are divided into a number of categories:

1. Arithmetic operators specify a register to register hardware instruction. As an example SUBX specifies the integer subtract instruction.

2. Addressing operators specify an address for use in load and store instructions.

   OFFS specifies an address as the address of some entity defined by a symbol table index with a fixed offset.

   SUBS specifies an address to be computed by adding a computed offset to an address defined by a symbol table entry or an OFFS instruction.

3. Load and store operators:

   LOAD translates into a load-instruction; its sole operand may be a symbol table index or may refer to an OFFS- or SUBS-instruction. Selection of a specific hardware instruction is left to the code-optimizer phase.

   REPL translates into a store instruction; it has a second operand designating the storee; otherwise it is similar to the LOAD-instruction.
   A special storee operand: "NONE", is used for optimization purposes. The code optimizer will not generate a store instruction for a REPL instruction with storee is "NONE", but it will use that REPL-instruction to suppress preceding store instructions into the same target. E.g. the code segment :

```
a:= b; statement-1 ; a:=c;
```

   where statement-1 designates some source valid source text, will give rise to two REPL-instructions with the same target and the code optimizer suppresses the first REPL-instruction, if "statement-1" allows it. In some cases e.g. at the closing of a range it is known that store instructions to local variables may be suppressed. E.g. the code segment :

```
(INT a; ... a:=b; statement-1)
```

   gives rise to a REPL-instruction for the variable a :

REPL a,b

for the closing of the range we add:

REPL a,NONE

the code optimizer suppresses the first REPL-instruction, if "statement_1" allows it. No code is generated for the second REPL-instruction.

4. Register designating operators:

SRV specifies that the result of the instruction specified by the first operand must be forced into the hardware register designated by the second operand. An SRV instruction is typically issued before a subroutine call.

DRV defines the content of the register specified by its sole operand. A DRV instruction is typically issued following a subroutine call.

5. Direct code operator:

DIRC specifies that the subsequent instructions, the number of which is specified by its second operand are Code File (CF) instructions to be copied without change to the output of the code-optimization pass. The first operand specifies in which 15-bit parcel of a 60-bit word the direct code may begin. DIRC is used to generate data entries in the code and also to generate code sequences for which the required or optimal register allocation is known.

6. Pseudo instruction operators:

LINE serves two purposes; Its first operand specifies the current source line number. A non-zero second operand tells the instruction scheduler in the code optimizer not to move instructions across this instruction. Typically the code for two consecutive units is separated by a LINE-instruction. Within a collateral clause the second operand will be zero and within a serial clause it will be non-zero. E.g. the ICF-code produced for:

(a:=b;c:=a)

and for:

(a:=b,c:=a)

will differ only in the second operand of the LINE-instruction separating the ICF-code for the two units.

EOS terminates the current sequence of instructions for the code optimizer.

TERM signals the end of the ICF.

3.6.5.2.5 Calling a FORTRAN routine.

For each call of a FORTRAN routine, a FORTRAN parameter list is generated. For data parameters the FORTRAN parameter list contains the address of the parameter, which is either the address of a copy of the value of the actual parameter or the address part of the actual parameter of mode REF to mode.

For a parameter of mode PROC, a mode independent FORTRAN callable interface routine is generated and the address of the generated routine is placed in the parameter list. As the code optimizer does not allow code motion, the interface routine is generated in the data section rather than the code section. Code is generated to store the actual parameter in the generated interface routine. The layout of the routine is described in 4.5.1.3.

3.6.5.2.6 Expand ICF-macros.

Upon an applied occcurrence of an operator defined by means of an ICF-macro, that ICF-macro is expanded with substitution of its formal arguments with the actual operands of the operator..

3.6.6 Code optimization and register assignment.

This phase processes a buffer of ICF-instructions at a time. The steps in the processing of such a buffer or sequence of instructions are discussed in the subsections below:

. Input of ICF and building of a dependency tree.
. Prioritizing the instructions.
. Register assignment.

3.6.6.1 Input of ICF and building of a dependency tree.

ICF (Intermediate Code File) instructions are input into an instruction buffer.

LOAD-instructions are inserted for operands that are symbol table indices. Common subexpressions are found; ICF-instruction are chained from their first operand; for each ICF-instruction entered into the buffer the chain from its first operand is scanned for an instruction with the same second operand. If such an instruction is not found, the new instruction is entered into the chain. If however such an instruction is found the new instruction is made to refer to that instruction.

The operands of the new instruction may become redundant. Redundant instructions are eliminated during the prioritizing step, described below.

Instructions directly following an unconditional branch instruction are unreachable. All reachable instructions read from ICF as well as the inserted LOAD-instructions are chained.

Some limited flow control analysis is performed; the sequence of instructions is divided in subsequences delimited by label or branch instructions. Two relations between successive subsequences A and B are established:

1. Subsequence B can be reached only from sequence A, in which case the result of instructions in A may be used in B.

2. Subsequence A must lead into subsequence B, in which case a REPL-instruction in A with the same target as a REPL-instruction in B may be eliminated.

3.6.6.2 Prioritizing the instructions.

Each ICF-instruction is given a priority. The priority of an ICF-instruction equals the execution time of the instruction plus the highest priority of any instruction using this instruction as an operand.
The chain of retained instructions created in the preceding step is scanned in a backward direction, thus in this scan the operands of an instruction are visited after the instruction itself. Thus instructions that are not (longer) referred to are easily found and eliminated.

Also if two REPL-instructions with the same target are found and the subsequence containing the first must lead into the subsequence containing the second, the first REPL-instruction, which is encountered last, is eliminated. Recall that in the code generator at range exit, REPL-instructions with dummy storees are issued for the variables declared in the range. These REPL-instructions will first cause other REPL-instructions to those variables to be eliminated and will later be eliminated themselves.

3.6.6.3 Register assignment.

This step determines the actual machine instructions to be generated for the load and store instructions. The chain of ICF-instructions resulting from the previous step is scanned and registers are assigned to them.

A table of register content descriptors is maintained. The content descriptor of a register contains the index of the ICF-instruction of which it is the result and possibly the index of the symbol table entry of an identifier of which the register holds the current value.
When a forward branch instruction is encountered the current register content table is placed in a chain of register content records associated with the target label.

When a label is encountered, the table of register content desriptors is initialized to "unknown" for all registers. If no backward branch to the label exists the register content records associated with the label are compared. If for some register all records show the same content the current register content of that register is set to that common content.

Registers are assigned for the result of each instruction in the following steps:

. If the instruction appears as operand of an SRV-instruction, we try to assign the target register specified in that SRV-instruction.

. Next we try to assign a register that holds one of the operands of the instruction.

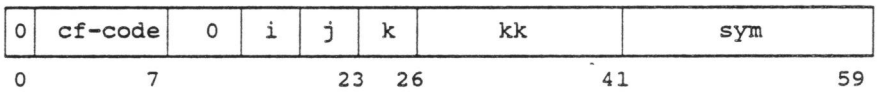
. Next we try to find a free register.

. If no free register is found a limited reordering of instructions is performed in an attempt to free a register.

. As a last resort a register is freed by forcing the content of some register to be stored in memory.

3.6.6.4 Output of CF (Code File).

The CF (Code File) contains one entry for each generated machine instruction. Each entry occupies one word in the format:



where,

cf-code represents the instruction-code,

i and j are the register indications for the operands of the instruction,

depending on the type of the instruction,
k indicates the result register or
k and kk together contain the address part of the instruction and

sym contains a symbol table index to allow printing of identifiers in the object code listing.

3.6.6.5 An example of optimization.

For the code fragment:

IF bool THEN statement-1 ELSE statement-2 FI

the code generated is equivalent to:

```
    IF NOT bool THEN GOTO else_label FI;
    statement-1;
    GOTO end_label;
else_label:
    statement-2;
end_label:
```

If statement-1 is a GOTO-statement the branch to end_label is unreachable and is removed, so we get:

```
    IF NOT bool THEN GOTO else_label FI;
    GOTO some_label;
else_label;
    statement-2;
    end_label:
```

The optimizer now changes this to:

```
    IF bool THEN GOTO some_label FI;
    statement-2;
end_label:
```

Further assuming that bool_expr has the form : i = j where i and j are of mode INT and that the operator "=" is defined by:

```
    OP = (INT, INT) BOOL =
    PR inline
          head  5,1
       a  subx  /1,/2      # a:= i-j;                                  #
          nz    r1,a       # IF a  <> 0 THEN GOTO else_label FI;       #
       b  subx  /2,/1      # b:= j-i;                                  #
       1  eqvx  a,b        # a*b ≥ 0                                   #
    PR  SKIP;
```

"a" and "b" are labels, that serve to reference the result of the labeled ICF-instructions.

The operand "r1" is defined only, if the boolean expression is used as selector in an IF or WHILE statement and designates the else- or false-label; an instruction containing "r1" as an operand where "r1" is not defined, is suppressed during expansion of the macro.

In our case however "r1" is well defined, but there is no reference to the result of the last two instructions and the optimizer deletes them.
Thus for the code fragment :

```
    IF i = j THEN GOTO some_label ELSE statement-2 FI;
```

the code generated corresponds to:

```
    a := i-j;
    IF a ≥ 0 THEN GOTO else_label FI;
    b := j-1;          #not referenced#
    a*b ≥ 0            #not referenced#
    GOTO some_label;
    GOTO end_label;    #unreachable#
else_label:
    statement-2;
end_label: ;
```

which is reduced to

```
    a := i-j;
    IF a = 0 THEN GOTO some_label;
    statement-2;
```

where the first two lines result in just two ICF-instructions:

```
a   subx   i,j
    zr     else_label,a
```

The production of this code requires hardly any optimization in the code-generator phase.

3.6.7 Editor.

The functions of the EDITOR are

. Output of diagnostics.
. Generation of a loadable module.
. Generation of object listing.

3.6.7.1 Output of diagnostics.

The diagnostic messages are classified by increasing severity, such as trivial, warning, error or fatal. A parameter of the compilation control statement specifies the lowest severity for which diagnostic messages are to be listed. Diagnostic messages are ordered first by compiler phase and within each phase by line number. The messages displayed for each phase are preceded by a message identifying that phase.

3.6.7.2 Data allocation and generation.

Appropriate loader tables are generated from the entries in the symbol table that represent the data sections of the program.

3.6.7.3 Generation of the object code.

The CF file is read and the instructions read are converted into loader table format. Addressing of data and labels in the program is finalized.

3.6.7.4 Generation of object listing.

If the compilation control statement parameter specifies listing of the object code, the allocation and initial content of data as well as executable instructions are listed in a format similar to that of the assembler.

4 RUN-TIME ORGANIZATION.

4.1 Control Data Cyber Memory and Representation of Data.

In this section the main characteristics of the memory organization and the internal data representation and its use in the ALGOL 68 system are described.

4.1.1 Memory organization.

In the Control Data Cyber computer system in use at the time of the ALGOL 68 project the main memory, also called small core memory or SCM, consists of a single one dimensional array of consecutively numbered 60-bit words. Addresses are 18 bits wide.

The memory available to a user program is limited to 2^{17} or 131072 words. Addresses used in a user program are positive 18-bit integers.

The 7000 series computers additionally contain a so-called large core memory or LCM, consisting of up to 2^{19} or 534288 60-bit words. Special instructions exist to move a word from a register to LCM and vice-versa.

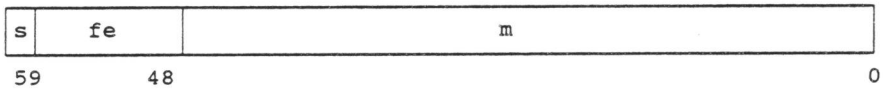
A special version of the ALGOL 68 compiler and run- time system was developed for the 7000 series. This product was called the LCM-version. The normal version was called the SCM-version. This document describes the SCM-version only. The implementation of the LCM-version is relevant only because it influenced some design decisions.

4.1.2 Internal data representation.

The internal data representation in the Cyber computer greatly influenced the design of the run-time system and of the garbage collector in particular.

In the Cyber computer integer and floating point numbers are represented in one's complement mode in full 60-bit words. Hardwarewise the range of integer values is $(-2^{59}+1, 2^{59}-1)$. The hardware integer multiply instruction operates on integers in the range $(-2^{47}+1, 2^{47}-1)$ only.

A floating point number represents a value: $2^{exponent} * mantissa$, where mantissa is an integer value in the range $(1-2^{48}, 2^{48}-1)$ and exponent is an integer value in the range $(1-2^{10}, 2^{10}-1)$. The internal representation of a floating point number consists of three fields designated s, fe and m :



where

s occupies the most significant bit and contains the sign of the mantissa.

fe occupies the second through twelfth bits and represents the exponent. The nature of its representation is described below.

m occupies the lower 48 bits and contains the lower 48 bits of the mantissa in one's complement representation.

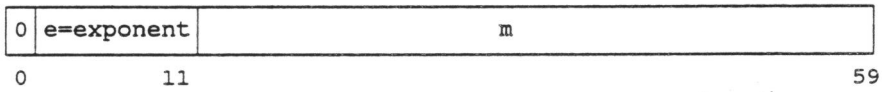
the concatenation of s and m is the 49-bit one's complement notation of the mantissa.

The fe field represents the exponent e. Starting from the 11-bit one's complement representation of the exponent the fe field is derived as follows: first the sign bit is complemented, then each bit is replaced by the result of an "exclusive or" operation on the bit and the sign of the mantissa.

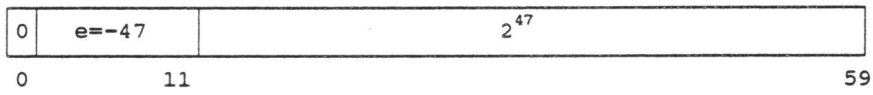
The hardware "PACK" and "UNPACK" instructions perform this transformation and the reverse transformation:

> PACK constructs a floating point number from the mantissa in its first operand register and the exponent in its second operand register.
>
> UNPACK accepts a floating point number as its sole operand and delivers two results: the primary result is the mantissa in 60 bits one's complement notation and the secondary result is the exponent of the operand in a 18 bit register.

In order to abstract from the internal representation of the exponent, we will in most cases show the true exponent value rather than the value of the fe-field. The format of a floating point number is shown as:



e.g. the value 1.0 would be shown as:



Three exponent values play a special role:

an exponent of 1023 or 2^{10}-1 designates an "infinite" floating point number,

an exponent of -1023 or 1-2^{10} designates zero (exponent underflow),

an exponent of -0 designates an "undefined" floating point number.

Any floating point number with an exponent of 2^{10}-1 is considered out of range or infinite and special hardware instructions exist to test floating point numbers for an out of range condition. Any hardware floating point operation yielding an out of range result produces a result with exponent 2^{10}-1and an m field of all zeroes.

Any floating point number with an exponent of -0 is considered indefinite and special hardware instructions exist to test floating point numbers for an indefinite condition. Any hardware floating point operation yielding an indefinite result produces a result with exponent 2^{10}-1 and an m field of all zeroes.

The table below shows some real numbers and their internal representations:

value	exponent	mantissa	octal internal representation
+0.0	-1023	0	0000 00000000000
+1.0	-47	2^{47}	1720 40000000000
-1.0	-47	-2^{47}	6057 37777777777
+2.0	-46	2^{47}	1721 40000000000
-2.0	-46	-2^{47}	6056 37777777777
+inf	+1023	0	3777 00000000000
-inf	+1023	0	4000 00000000000
+indef	-0	0	1777 00000000000
-indef	-0	0	6000 00000000000

Thus as for integer numbers the internal representation of a negative floating point number is the one's complement of the internal representation of the absolute value of the number. Also if a bit pattern represents a larger floating point number than another bit pattern, it also represents a larger integer value than the other bit pattern.

4.1.3 ALGOL 68 use of the internal data representation.

The term pointer is used for every value that refers to some other value. The two most frequently used kinds of pointers are:

(1) values of mode REF-to-Mode or references

(2) static parts of multiple values. These contain the bound information and the address of the first element of the multiple.

The design of the run-time system and in particular of the garbage collector is based on the distinguishability of pointers from non-pointer values.

Values of mode INT are restricted to the range $(-2^{47}+1, 2^{47}-1)$.

Values of mode REAL are internally represented as a non-infinite non-indefinite floating point number or as an infinite or indefinite floating point number with an m-field consisting of all zeroes.

All other simple plain values, i.e. values of type BOOL, CHAR, BITS or BYTES are represented by a full 60-bit word of which the 12 most significant bits are all zeroes.

Bit patterns that represent an out of range floating point value and show a non-zero mantissa are used exclusively for the representation of pointers.

Values that do not contain pointers are termed plain values.

4.2. Memory Organization.

The internal representation of ALGOL 68 values as well as the allocation of data and initialization of the stack and the heap are designed to comply with the stringent requirements of the run-time system and of the garbage collector in particular.

The paramount requirement is the distinguishability of pointers and plain values. Therefore all allocation areas must be initialized to plain values.

This section discusses:

. Storage allocation of ALGOL 68 values.
. The run-time stack.
. The heap.
. Scope checking.
. Garbage collection.

4.2.1 Storage allocation of ALGOL 68 values.

The storage allocated to a value consists of one or two parts, called the direct and the indirect part respectively. The size of the direct part depends on the virtual mode of the value and is thus static and known at compile time.

The size of the indirect part depends on the actual mode of the value and is therefore not necessarily known at compile time. In this implementation the size of the indirect part is always treated as dynamic, i.e. no attempt is made to evaluate the size of the indirect part at compile time.

Values for every mode have a direct part.

Values of mode PLAIN, PROC, REF and UNION consist of a direct part only.

Multiple values consist of a direct part and an indirect part. The direct part contains the bound information and a pointer to the indirect part. The indirect part consists of the elements of the multiple.

Structured values consist of a direct part, derived from the direct parts of the components by concatenation in order of definition, and an indirect part consisting of the indirect parts, if any, of its components .

In this implementation elements of a multiple as well as objects of references are always allocated on the heap. A number of reasons can be identified for this allocation method, of which two are given here and some more in the next section.

The first reason is that as mentioned in section 2.1.3 allocation on the heap of the objects of references obviates run-time scope checks, otherwise required in cases where the scope of the reference cannot be determined at compile time.

The second reason is that on the Control Data 7000 systems the heap is allocated in LCM rather than in SCM. Since access to LCM and SCM is done by different hardware instructions, the storage class of each object must be be known at compile time.

4.2.2 Run time stack.

Local or automatic data is allocated on a run-time stack that contains at any point in time a stack segment for every active range. Contiguous allocation of stack segments in a single memory segment in the order of range nesting minimizes the memory required for the storage of local data.

Apart from ALGOL 68 values the run-time stack contains control information such as routine- and range-headers and FORTRAN parameter-list descriptors. The garbage collector examines every

single word of the run-time stack and therefore the bit patterns for these items must be distinguishable from pointers.

For simplicity of stack handling all allocation on the run-time stack is defined at compile time. The size of the stack segment for every block is fixed by the compiler. This is another reason that the elements of a local multiple are not allocated on the run-time stack but on the heap.

When calling a procedure we only have to advance the stack pointer by a fixed amount and when the callee has returned the caller only needs to move the stack pointer backwards by the same amount. Thus the so-called dynamic link is not materialized in this implementation.

Also some run-time routines that need to perform a (partial) trace back determine the stack pointer of the next outer routine activation by applying the offset found in the instruction that lowers the stack pointer after a call.

4.2.3 Heap.

Global data i.e. data with unlimited life time or scope, is allocated in ALGOL 68 by a so-called global generator which yields a pointer to the newly allocated data. The data item is said to be the object of the pointer. The area in memory containing the dynamically allocated data is conventionally called a heap.

In this implementation the heap also contains values that have a limited lifetime but that we do not wish to allocate on the stack, e.g. because the size of the value is not static or its use may otherwise necessitate run-time scope checking.

4.2.4 Scope checking.

In ALGOL 68 the object of a pointer may be local as well as global. If a pointer to a local object allocated on the stack is assigned to a pointer or variable with a longer life time than the pointer's object, then upon expiration of the object's life time the pointer continues to exist but its object does not.
This phenomenon is called the dangling pointer problem.

ALGOL 68 scope rules prohibit such assignments by demanding that in an assignment the scope of the source must include the scope of the target, or in other words, the life time of the source must at least be equal to the life time of the target.

For some assignments the scopes of source and target cannot be determined at compile time. Rather than to incorporate a run-time scope check in the object code the system treats the scope of the source as global and allocates it on the heap. This approach results in a simplification of the compiler and the generated

object code at the expense of more data being allocated on the heap. Note that it is not a deviation from the ALGOL 68 standard but rather a superlanguage feature since it assigns a sensible meaning to an action the result of which is called undefined by RA68.

The scope of routines causes a different problem. Forcing all data accessible through a routine to be allocated on the heap would imply changing the allocation of most data from the stack to the heap. On the other hand the accessibility of data through a routine remains indirect until the routine is invoked. We therefore decided to check the scope of a routine upon its activation rather than upon assignation, ascription, etc. as implied in RA68.

4.2.5 Garbage collection.

Generally the global data in the heap are accessible through pointers only. Although such data may have unlimited lifetime, the lifetime of the pointers to it is not necessarily unlimited.

When the lifetime of all pointers giving access to a particular piece of global data have expired, that piece of data becomes inaccessible.

As the ALGOL 68 language has no explicit deallocation statement, a mechanism for deallocation is needed to retrieve the memory containing inaccessible data.

The process that retrieves the memory allocated to objects to which no pointer exists is called garbage collection.

In ALGOL 68 the garbage collector problem is complicated by the fact, that the object of a pointer may consist of a number of parts that are allocated in disjoint memory segments.

ALGOL 68 includes operations on pointers that yield a pointer to part of the data referred to by the argument pointer. An example is the field selection applied to a value of mode REF STRUCT.

The objects of two pointers may overlap without one object being contained in the other, e.g. after the declaration:

```
[1:10] REAL r10;
```

the objects of the pointers: r10 [1:5] and r10 [3:8] overlap.

Also the object of a pointer may consist of a number of non-contiguous parts. E.g. after the declaration:

```
[1:10, 1:10] INT i_10_by_10;
```

the object of the pointer: "i_10_by_10 [2:3, 2:4]" consists of the 6 elements:

i_10_by_10[2,2], i_10_by_10[2,3], i_10_by_10[2,4],

i_10_by_10[3,2], i_10_by_10[3,3] and i_10_by_10[3,4] ,

which will be allocated in two contiguous segments of three elements each.

As any type of data may be dynamically allocated the object of a pointer may contain one or more pointers.

The task of the garbage collector is to make the memory vacated by data no longer reachable through existent pointers available for allocation of data.

As this process in the general case frees a number of disjoint memory segments the garbage collector must be able to move the data to be retained into a contiguous memory segment in order to avoid memory fragmentation and to adjust all pointers referring to the moved items accordingly.

As a FORTRAN routine having access to data allocated by the ALGOL 68 system may remain active across an activation of the garbage collector, the movement of data by the garbage collector causes a problem for the access of such data by the FORTRAN routine.

This problem occurs when a FORTRAN routine, directly or indirectly called by the ALGOL 68 program, calls an ALGOL 68 procedure that activates the garbage collector.

The code produced by the FORTRAN compiler handles addresses of parameters in many different ways and therefore parameters of FORTRAN routines must not be moved as long as the FORTRAN routine remains active.

As the design of the FORTRAN interface restricts access to ALGOL 68 routines from a FORTRAN routine to procedural parameters of the FORTRAN routine, the problem applies only to the activation of FORTRAN routines possessing at least one procedural formal parameter.

When a FORTRAN routine is called from an ALGOL 68 procedure and that routine has one or more formal procedural parameters, a parameter-list descriptor is stored on the run-time stack. Thus for every activation of a FORTRAN routine from the ALGOL 68 program a parameter-listdescriptor exists in the run-time stack.

The garbage collector recognizes these descriptors, determines the addresses of all data allocated by the ALGOL 68 system and accessible to a FORTRAN routine and ensures that such data will not be moved.

As FORTRAN parameter lists contain only the starting addresses of the parameters, the solution adopted is not to move any data allocated at addresses higher than the address of any data

accessible to a FORTRAN routine.

The treatment of the scope rule and local arrays tends to increase the amount of data allocated on the heap and hence of garbage collector activity.

In summary the Cyber ALGOL 68 garbage collector must be capable to handle pointers to any size objects with no restrictions on overlapping of objects, to perform data compaction and pointer adjustment and execute very efficiently.

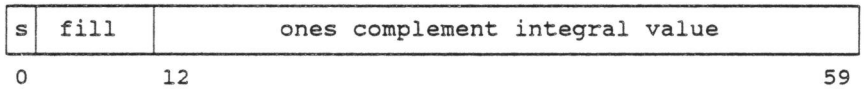
4.3 Internal Data Representation.

All data are represented in an integral number of 60-bit words. ALGOL 68 values are categorized in:

. Simple plain values.
. Simple non-plain values.
. Structured values.

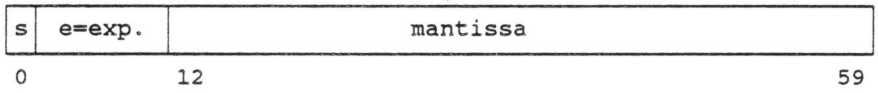
4.3.1 Simple plain values.

INT

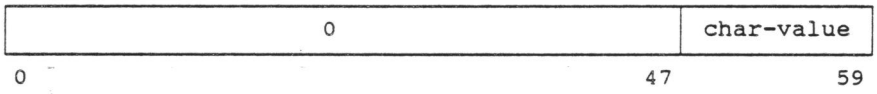


The fill bits in positions 1 through 11 are equal to the sign of the integer value located in bit 0.

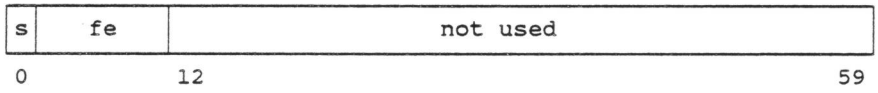
REAL



CHAR

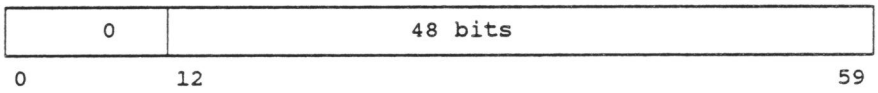


BOOL

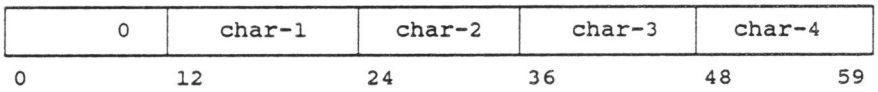


where
the exponent represented by s and e is not equal to 1023,
s = 0 for FALSE, s=1 for TRUE.

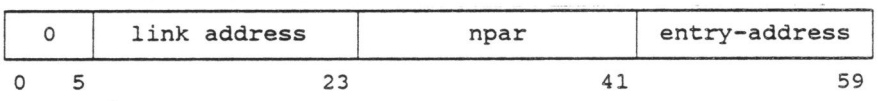
BITS



BYTES

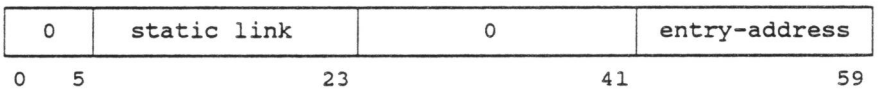


PROC (not requiring scope check)

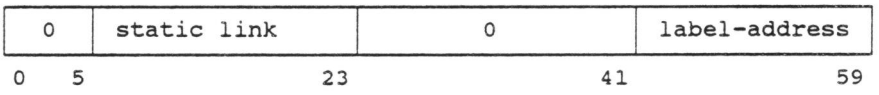


This is the general pattern. A PROC-word represents an ALGOL 68 routine-text, a procedurized label or a routine written in some other language, the particular representations of which are shown below. An ALGOL 68 routine may or may not require scope checking. Here we show the internal representation of a routine-text not requiring scope checking.

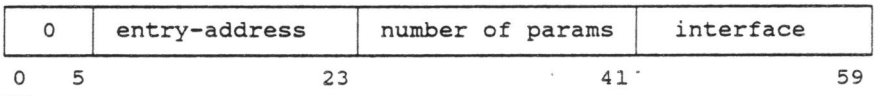
routine text



label



other language routine



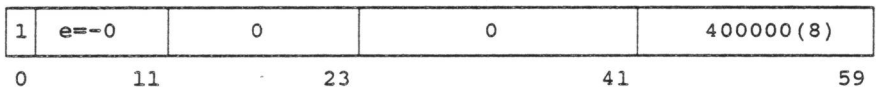
where

entry-address is the entry point address of the other language routine,

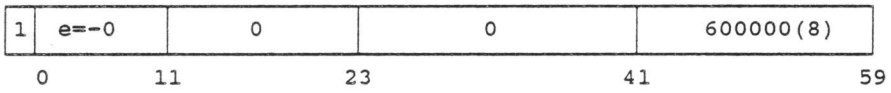
numpar is the number of parameters and

interface is the entry-point address of the interface routine, which adapts the parameter list to the language in which the other language routine is written.

SKIP

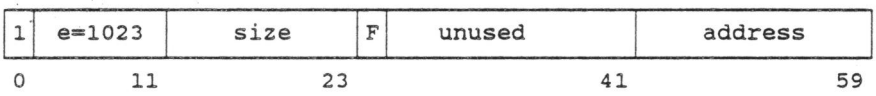


NIL

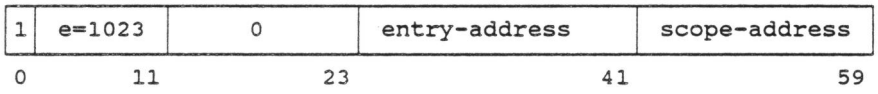


4.3.2 Simple non-plain values.

REF amode.



PROC (requiring scope check)



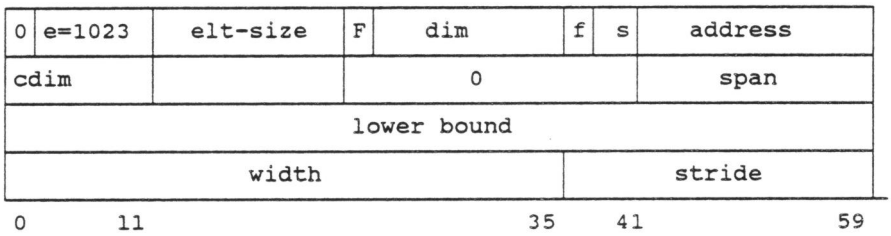
4.3.3 Structured values.

STRUCT

The internal representation of the direct part of a value of mode STRUCT consists of the concatenation of the internal representations of the direct parts of its fields, in the order of definition.

ROW-of-Mode

The internal representation of a multiple consists of two parts: the static part contains the multiple-descriptor. The dynamic part contains the elements of the multiple. The multiple descriptor of an n-dimensional multiple occupies 2+2*n words in the following format:



where the last two words appear once for each i-th (i= 1, n) dimension and

```
elt-size    = size of element - 1,
F           = mark-flag: F=0 : element is a plain value,
                         F=1 : element may contain pointers,
dim         = number of dimensions = n,
f           = flexible-flag,
s           = reserved,
c           = compact flag: c=0 : multiple is compact,
                            c=1 : multiple is  not compact,
address     = address of first element,
cdim        = compactification metric,
span        = size of multiple,
lower bound = lower bound[i],
width       = upper bound[i] - lower bound[i] + 1 and
stride       = distance of two elements with i-th subscripts
               one apart.
```

Cdim represents the number of outer loops required to access all elements of the multiple and equals zero for a compact multiple A declaration always yields a compact multiple, but a slice often does not.
Cdim = dim is possible only for a slice that results from selection.

Span is the length in words of the smallest segment of memory containing all elements of the multiple.

E.g.after:

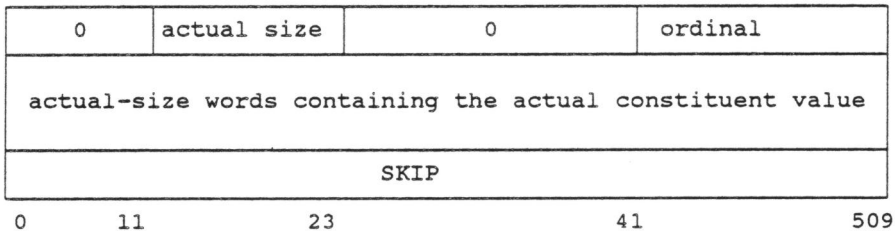
```
[5,7,9] COMPL arr3;
```

the descriptors of the multiples shown below are as follows:

slice	c	cdim	span
arr3	0	0	5*7*9*2 = 630
arr3 [2:4, , ,]	1	0	3*7*9*2 = 378.
arr3 [2:4,3:6,]	1	1	3*7*9*2 - (7-4)*9*2 = 324.
arr3 [, , 5:8]	1	2	5*7*9*2 - (9-4)*2 = 620.
re OF arr3	1	3	5*7*9*2 - (2-1) = 629.

The cdim and span fields are included for the benefit of the garbage collector and the run-time routines for dereference and assignation of multiples.

UNION



where

actual-size is the size of a value of the actual mode and

ordinal is the ordinal associated with the actual constituent mode.

The number of words allocated is one more than required for any constituent mode.

The unused space beyond the actual value is called the residue. All words of the residue contain SKIP values.

4.4 Memory Layout.

The lower part of memory contains the ALGOL 68 object program code as well as the necessary run-time routines and possibly the code and local data of procedures written in other languages.

The run-time stack is allocated direct beyond the program. During execution of the program the stack will grow and shrink as program blocks are activated and deactivated.

The heap is a contiguous segment of memory allocated at the top of available memory and growing downwards as objects are allocated on the heap.

The memory segment between the top of the stack and the bottom of the heap is used to allocate stack segments and new heap objects. When this segment is exhausted the garbage collector is activated to compact the accessible data in the heap upwards and thus create allocatable space between stack and heap.

In case not enough memory space can be freed by the garbage collector an operating system request is issued to increase the amount of memory available.

Memory allocation is done in full 60-bits words exclusively.

All memory allocated in the stack, the heap and the area designated for extension of the heap is preset with SKIP-values.

A detailed memory map is given in section 4.4.4.

4.4.1 Run-time stack.

A stack segment of a routine consists of a procedure-segment and a range-segment for each active range. The maximum length of a routine-segment is a compile-time constant. When a routine is invoked a segment of this maximum length is allocated.

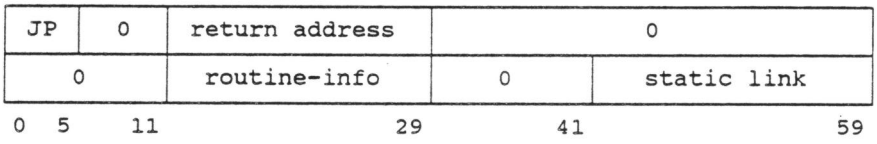
At any point in time a routine-segment contains the routine-header and the range-segments of the active ranges in the routine.

The length of the segment actually used therefore varies during execution of the routine. When another routine is called however, its stack segment is allocated above the actually used part of the caller's stack segment, rather than above the stack segment initially allocated for the caller.

This arrangement not only economizes stack pace, but it also simplifies the run-time system especially the garbage collector.

4.4.1.1 Procedure-segment.

A procedure-segment occupies two consecutive words in the format:



The first word is called the return information, in which "JP" stands for the instruction code of an unconditional branch and "return-address" is the address immediately following the

subroutine jump. This word is generated by the hardware when a subroutine jump is executed.
The return from a procedure is effected by a branch to the return-information.

The second word is called the procedure header in which the field "routine-info" contains the address of the statically allocated routine-information entry and the field "static link" contains the stack pointer of the active procedure containing the range that is the scope of the routine.

There is no so-called display in the stack. Data in stack segments of surrounding routine activations are reached by following the chain of static links in the stack until the stack segment of the target activation is reached. Obviously the number of links to follow is known at compile time.

The number of links to follow is thought to be relatively small. The address of the outermost stack segment is in a dedicated register and stack segments that are not referred to from within the procedure do not appear in the chain of static links.

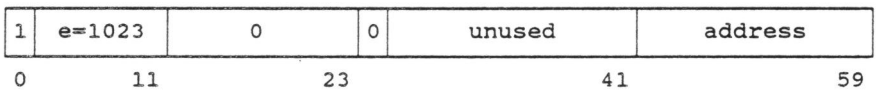
4.4.2 Range-segment.

A range-segment may contain a range-header, a static identifier stack or Sid and a static working stack or Swo. The terms "Sid" and "Swo" were invented by Branquart [20].

A parameter range contains no operator or mode declarations and no executable code. A parameter-range segment does not contain a Swo. The range-header, if any, of a parameter-range-segment is allocated after the Sid, because the position of the Sid of the parameter range immediately following the procedure-segment of the callee is dictated by the caller-callee interface.

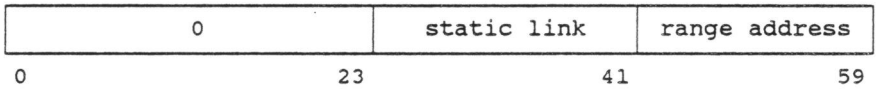
4.4.2.1 Range-header.

The range-header exists only for ranges that are the scope of some routine and is used only to check the scope of a routine at the time the routine is called.
A range-header consists of a single word in the format of a reference to a single word on the heap. On each entry into the range a new word on the heap is requested. A range-header thus uniquely identifies the activation of the range. The format of the range-header is:



The *scope-word* in the heap referred to by the field "address" represents the activation of the range. It has two fields

containing the address of the range-header in the stack and the address of the routine-header of the routine containing the range. This address equals the static link of every routine, of which the range is the scope. The format of the scope-word is:



4.4.2.2 Static identifier stack.

The Sid is of fixed size and contains the direct parts of allocatable entities declared in the range. The entities to be allocated may result from an identity declaration, an operation declaration or a mode declaration.

An identity declaration that is not a variable declaration, results in a value of the specified mode to be allocated.

A variable declaration results in a reference to a value of the specified mode in the Sid or if the variable is "optimized" in a value of the specified mode in the Sid. (optimization of variables was described in the section on the semantic scan. Here it may suffice to state that a variable can be optimized, if its scope can be checked at compile time).

An operation declaration gives rise to an entry in the stack if it defines the operator in terms of ALGOL 68 . When however the operator is defined in terms of ICF-instructions by means of a so-called ICF-macro no entry in the stack is allocated for the operator.

An operation declaration not using an ICF-macro actually defines a procedure. It differs from a procedure declaration in that it does not associate an identifier with the routine it defines. Thus the routine cannot be manipulated as a value and an operator can therefore never be activated out of scope and the entity in the Sid has the format of a PROC-word not requiring scope check.

A mode definition gives rise to a PROC-word in the Sid, if a value of the specified mode has an indirect part, i.e. if the mode "shows" a multiple.
The routine generated for such a mode declaration has two parameters: the number of values of the specified mode to be generated and the address, where the generated values must be stored. As with an operator definition the resulting PROC-word is of the "no check required" type.

Values that are recognized to be load-time constant are not allocated in the Sid.

A PROC-word of a routine with global scope has its static link set to zero and constitutes therefore a load time constant.

The PROC-word of a non-global routine contains a non-zero static link. It could of course be constructed when needed but then a number of links in the chain of static links might have to be accessed. Therefore we decided to generate the PROC-word at its point of definition and store it in the Sid.

4.4.2.3 Static working stack.

The static working stack or Swo contains i.a. operands of expressions being evaluated and parameters of procedures to be called. If the routine being called is a FORTRAN routine possessing at least one parameter of mode PROC, the Swo contains also a so-called FORTRAN parameter-list descriptor in the format:

0	e=1023	number of parameters	1	address

The size of the Swo varies during the elaboration of the range, but its actual size is always known at compile time.

The size of the stack segment for an active routine varies during the elaboration of the routine. When the routine calls another routine the size of its stack segment cannot change until the callee returns control to the calling routine. The size of the stack segment of an active routine is known at compile time.

Therefore the run-time stack never contains unused segments. The top of the active stack can always be computed from the address of the currently active routine's stack segment that is kept in a dedicated hardware register and the current size of that segment. The determination of the top of the allocated stack however requires a special subroutine described in section 4.5.3.3.

4 4.4. Memory layout table.

<table>
<tr><td>PROGRAM</td><td colspan="2">ALGOL 68 main program
ALGOL 68 separate routines
Library routines
Other language routines
Constants</td></tr>
<tr><td rowspan="2">B3=>
program
segment</td><td colspan="2">global information</td></tr>
<tr><td colspan="2">procedure header</td></tr>
<tr><td>range-</td><td colspan="2">range-header</td></tr>
<tr><td rowspan="3">segment
of
standard
circumlude</td><td rowspan="3">Sid
or
static
identi-
fier
stack</td><td>mode-descriptors</td></tr>
<tr><td>identity-descriptors
routine-descriptors
variables</td></tr>
<tr><td>operator-descriptors</td></tr>
<tr><td rowspan="3">range-
segment
of
user
circumlude</td><td colspan="2">range-header</td></tr>
<tr><td rowspan="2">Sid
or
static
identi-
fier
stack</td><td>mode-descriptors</td></tr>
<tr><td>identity-descriptors
routine-descriptors
variables</td></tr>
<tr><td colspan="3">operator-descriptors</td></tr>
<tr><td colspan="3">range-segment(s) of user circumlude(s)</td></tr>
<tr><td rowspan="4">range-
segment
of outer
range
of main
program</td><td colspan="2">range-header</td></tr>
<tr><td rowspan="2">Sid
or
static
identi-
fier
stack</td><td>mode-descriptors</td></tr>
<tr><td>identity-descriptors
routine-descriptors
variables</td></tr>
<tr><td>operator-descriptors</td></tr>
<tr><td>Swo</td><td>intermediate results</td></tr>
<tr><td colspan="3">range-segment(s) of inner ranges of main program</td></tr>
</table>

<table>
<tr><td colspan="4">procedure-segment(s), actual parameters and
range-segment(s) of active procedures</td></tr>
<tr><td rowspan="2">procedure
segment</td><td colspan="3">return-information</td></tr>
<tr><td colspan="3">procedure header</td></tr>
<tr><td rowspan="2">parameter
range-
segment</td><td colspan="3">Static identifier stack</td></tr>
<tr><td colspan="3">range-header</td></tr>
<tr><td rowspan="3">range-

segment</td><td colspan="3">range-header</td></tr>
<tr><td colspan="3">Static identifier stack</td></tr>
<tr><td colspan="3">intermediate results</td></tr>
<tr><td rowspan="6">range-

segment</td><td colspan="3">range-header</td></tr>
<tr><td colspan="3">Static identifier stack</td></tr>
<tr><td rowspan="4">Swo</td><td colspan="2">intermediate results</td></tr>
<tr><td colspan="2">0</td></tr>
<tr><td colspan="2">routine-descriptor of callee</td></tr>
<tr><td colspan="2">actual parameters</td></tr>
<tr><td colspan="4">TOP=> allocated stack space</td></tr>
<tr><td colspan="4">TAS=> available for stack extension, content undefined</td></tr>
<tr><td colspan="4">B4=> available for heap extension, contains SKIP-values</td></tr>
<tr><td colspan="4">B2=> H E A P</td></tr>
<tr><td colspan="4">hole</td></tr>
<tr><td colspan="4">RFL=></td></tr>
<tr><td colspan="4">MRFL=> requestable memory</td></tr>
</table>

NOTES

1. The ra n ge-segments of the standard circumlude and user circumludes are all included in the procedure-segment of the main program.

2. Rang e -segments of circumlude-ranges do not contain any intermediate results (no Swo-part) because the main program is not contained in an expression.

3. Variables declared in a circumlude are not optimized, because their usage is unknown at compile-time and such variables may be accessed not only from a main program but also from separately compiled routines.

4. The hardware B-registers B1,B2, B3 and B4 are dedicated as follows:

   B1: address of stack segment of innermost active routine
   B2: first word address of the (currently allocated) heap
   B3: address of the stack segment of the main program.
   B4: address of the padded area designated for extension of the heap.

   These registers contain the addresses indicated by the position of "B1=>",...,"B4=>" in the left margin.

   The further entries in the left margin are :

   "TOP=>" : indicates the top of stack,
   "TAS=>" : indicates top of allocated stack, see 4.5.3.3,
   "RFL=>" : indicates the top of memory allocated to the program,
   "MRFL=>": indicates the maximum of memory that can be obtained.

5. The table shows the content of the run-time stack during preparation of a call to a procedure. The Swo of the most recently activated routine at the bottom of the table shows after some unspecified intermediate results :

   . A zero-word and the routine-descriptor of the routine being called which will be changed into a procedure-segment by the procedure prologue code.

   . The actual parameters.

   Parameters are thus addressable from the callee as well as from the caller by offsetting the current stack pointer with a constant offset.

6. The area designated for extension of the heap is allocated above the stack segments of all active routines. Note that while the the actually used stack segments of active routines obviously must be distinct, a portion of a stack-segment allocated to some routine may be temporarily used for a routine called from the first routine.

4.5 Object Code Organization.

In this section we discuss:

. Object module layout.
. Code sequences for some specific language constructs.

4.5.1 Object module layout.

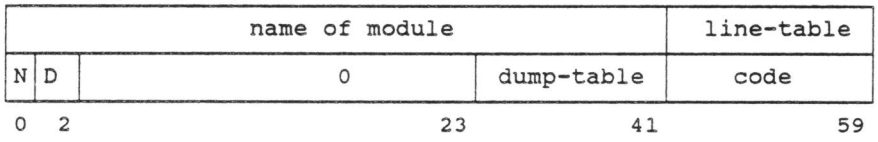
The object module contains:

. Module-information,
. Constants,
. FORTRAN mode dependent interface routines,
. Routines.

4.5.1.1 Module information.

The module information consists of the module-descriptor, the line-table and possibly a symbolic dump table.

The module-descriptor is a two-word entry in the format:



where the fields are used as follows:

name of module : a name of at most seven characters,

line-table : the address of the line table,

N : a two bit designator for the type of compilation:
N = 0 separate compilation,
N = 1 main program,
N = 2 circumlude,

D : a single bit reflecting the parameter for the generation of a symbolic dump table on the compilation-control statement:
D = 0: no dump table specified.
D = 1: dump table specified.

dump-table : if D=1 the address of the symbolic dump table otherwise 0.

code : the address of the first executable instruction in the module.

The line table serves to determine the source line number from a relative address in the code of the module. The line table is used only in the formation of run-time error messages.
The symbolic dump table provides the identifiers used in the program for the symbolic post mortem dump.

4.5.1.2 Constants.

The constant section of the module contains the constants used in the program that cannot be generated by means of one or two hardware instructions as well as compiler generated constants.

For STRING-denotations a multiple descriptor and the elements of the string exist in the constant section.

Compiler generated constants include mode independent FORTRAN interface routines, of which the COMPASS equivalent reads:

```
mifitr    DATA      0          entry/exit
+         SA1       A68rout    fetch a68 callee
-         RJ        mdfitr     convert parlist and call A68 rout
          EQ        mifitr     return to fortran routine
A68rout   BSS       1          holds A68 PROC-word
```

The format of a mode independent FORTRAN interface routine is:

JP	return-address	0	
SA1	par-PROC-word	RJ	mdiftr
EQ	mifitr	0	
par-PROC-word			

return-address = return point in the calling FORTRAN routine.

par-PROC-word = the PROC-word of the actual parameter, which is an ALGOL 68 procedure of the appropriate mode.

mdiftr = the mode dependent FORTRAN interface routine.

mifitr = the entry-point address of this interface routine.

4.5.1.3 Mode dependent FORTRAN interface routines.

For every PROC-mode occurring as the mode of a parameter of a FORTRAN routine a mode dependent FORTRAN interface routine exists. Its purpose is to convert the parameter list generated by

the FORTRAN compiler into a parameter list acceptable to an ALGOL 68 routine of the appropriate mode.

It serves as the bridge between the mode independent interface routine that was passed to the FORTRAN routine and the ALGOL 68 routine that was the actual parameter in the call to FORTRAN.
Depending on the mode of the ALGOL 68 callee, it transfers control to the descriptor-code associated with the callee or to the normal entry-point of the callee.

4.5.1.4 Routines.

The code produced consists of the descriptor-code, the routine-info and the executable code generated for the routine text.

The descriptor-code exists only for routines that may be called from a FORTRAN routine. It serves to compute the bound information for array parameters of which FORTRAN passes the address only and to pass control to the entry-point of the procedure.
The descriptor-code is generated from the descriptor-pragmat, which was described in section 2.5.4.4, and is accessed via the mode dependent FORTRAN interface routine only.

The routine-info is a two word constant in the format:

0	length of result	parlist-length	stack-size
source line of routine		0	module-info

where

parlist-length is the total length of the parameters,

stack-size is the maximum size of the routine-segment,

module-info is the address of the module-information entry.

The executable code for the routine contains at least the routine prologue and the routine exit code.
The routine prologue consists of a call to the run-time subroutine G;prol with the routine-info as parameter.
The routine exit code forces the result of the routine into the hardware registers, as required and returns control to the caller via the return-information in the run-time stack.

4.5.2 Code sequences for particular language constructs.

4.5.2.1 Procedure calling.

When a routine is called a preliminary routine header, consisting of a filler word and the PROC-descriptor of the callee, and the parameters are stored at the current top of the current routine-segment. The run-time subroutine G;call is called to activate the callee.

G;call checks the scope of the routine being called as required and transfers control to the callee.

The callee calls the routine prologue subroutine, which transforms the preliminary routine-header into a normal routine-header, redefines the current stack-segment by setting a dedicated register to point to the new header and allocates the new routine-segment on the run-time stack.

The parameters of the routine can now be accessed by offsetting the new routine-header with a fixed offset.

4.5.2.2 Range entry.

If this is a scope range a heap request for the scope-word is issued and a reference to it is stored on the run-time stack to serve as range-header.

The Sid is initialized to SKIP-values.

The heap space required for the direct parts of all non-optimized variables is requested by means of a single heap request.

4.5.2.3 Variable declarations.

Four cases are distinguished:

1. Optimized variable(s) with a non-multiple mode.

   Such variables are allocated on the run-time stack at compile-time. No code is required for the elaboration of the declaration.

2. Optimized variable(s) of a mode that shows one or more multiples.

   The static parts are allocated on the run-time stack at compile time. For each component that is of mode ROWS, the static part contains a multiple-descriptor.

   For each component of mode ROWS, the following four steps are taken:

- The dim and span fields in the descriptor header of the multiple in the first variable are computed.

- The bounds of all the component multiples are elaborated. The lower-bound, width and stride fields for all dimensions are computed and stored in the descriptors of the multiple components of the first variable declared.

- The number of variables is multiplied by the number of elements per multiple and the size of the element in words to yield the total number of words required for the elements of the multiples.
  The number of words is stored in the span-field of the descriptor of that component multiple of the first variable.
  A heap request for this number of words is issued and the resulting pointer is stored in the descriptor-header of that component multiple in the first variable.
  At this point the descriptors of the component multiples of the first variable act as references to all elements of the component multiples. This is necessary, because the next step may cause an invocation of the garbage collector.
  If an element again contains one or more multiples, the process is repeated.

- A run-time routine is called to initialize the elements. The first element is copied with the exception of the span-fields, which are set to reflect the number of elements of one multiple, and of the address fields that are set at the address of the preceding multiple plus the total size of the elements of one multiple.

3. Non-optimized variable with a non-multiple mode.

   A reference to part of the heap area requested during range entry processing is stored in the run-time stack.

4. Non-optimized variable with a mode showing one or more multiples.

   A reference to part of the heap area requested during range entry processing is stored in the run-time stack.

   The descriptors in the heap are initialized as described under item 2 above.

4.5.2.4 Parallel execution.

As mentioned in section 3.6.3.6 a parallel clause is turned into a call of the ALGOL 68 system routine Execpar with a parameter of the mode

```
[ ] STRUCT (PROC VOID x, INT y)
```

where each x-field represents one of the units in the parallel clause and the y-fields provided are used within Execpar to hold information on the unit and are initialized to zero.

4.5.2.5 Calling a FORTRAN routine.

A FORTRAN parameter list is allocated on the stack. Each parameter is elaborated, placed on the top of the run-time stack and converted into a FORTRAN parameter that is stored in the FORTRAN parameter list.
Conversion to a FORTRAN parameter is done as follows:

. For a parameter of mode PRIMOD a copy of the parameter is placed on the stack and the address of the copy constitutes the FORTRAN parameter.

. For a parameter of mode CHAR or [ ] CHAR the value is converted into the 6-bit code used in FORTRAN and placed on the stack. The address of the copy constitutes the FORTRAN parameter.

. For a parameter of mode REF PRIMOD, the address of the PRIMOD value is transmitted as parameter to FORTRAN.

. For a parameter of mode [] PRIMOD or REF [ ] PRIMOD the address of the first element is transmitted.

. For a parameter of mode PROC a mode independent FORTRAN interface routine is generated as a constant, in which the mdifr-fields points to the mode dependent FORTRAN interface routine for the mode of the parameter. The address of the constant constitutes the FORTRAN parameter.

4.5.3 Run time library routines.

The transput routines have been written in ALGOL 68 except the transput primitives. All other run-time library routines are written in the assembler language COMPASS.

The functions listed below are executed through library routines:

. Calling an ALGOL 68 routine.
. Routine prologue.
. Computing the top of stack.
. Parallel execution.
. Garbage collection.
. Transput.

4.5.3.1 Calling a routine.

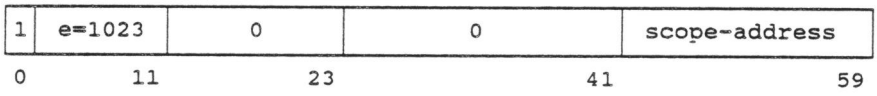
The subroutine G;call is called for each call of an ALGOL 68 procedure. Its input is the address in the run-time stack of the PROC-word of the ALGOL 68 routine to be activated.

The hardware stores a branch instruction to the return address at the entry point of G;call. G;call stores this instruction in the run-time stack immediately preceding the PROC-word.

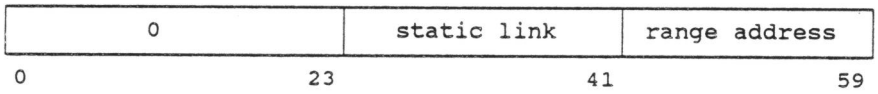
If the PROC-word is of the "no-check-required" type control is transferred to the callee.

Otherwise the scope of the routine is checked as described below.

Upon entry of a range that is the scope of some routine, a new word is allocated on the heap and a reference to that word is stored in the range-header of the range in the run-time stack.
This is a reference to a one word plain object and hence has the format:

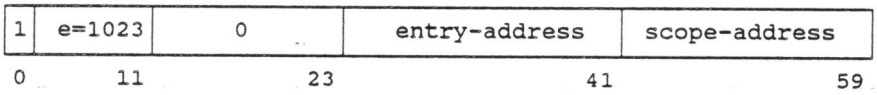


Thus for each activation of a scope range a unique word is allocated on the heap in the format:



where static link is the address of the current procedure-segment and range-address of the range-header word in the run-time stack.

The reference to this word uniquely identifies the activation of the range. Since all data in the stack are defined in this implementation, exactly one range-header is present in the stack for each activation of a scoperange. The descriptor of a routine contains a copy of the range-header uniquely identifying the associated activation of its scope range.



When the routine is called, the associated activation of its scope range should exist and thus the stack should contain the range-header corresponding to that activation at its original address.

The scope check on a routine being called consists of two parts: first the address of the range-header from the heap word referred to by the routine descriptor is compared to the address of the top of the stack.
If the check fails the range being the scope of the routine cannot be active, the routine is out of scope and the program is

aborted with an appropriate message.

Here the format of a value of mode PROC that requires scope checking is repeated for convenience:

1	e=1023	0	entry-address	static link

Otherwise, if the scope range is active, its header in the stack should be located at the address found in the heap-word and contain a reference to the heap-word.
This can be checked because the internal representation of a reference differs from the internal representation of any other type of value.
As the addresses yielded by distinct heap generators are distinct, the range-header represents the activation of the range associated with the routine, if and only if the heap-address in range-header and routine-descriptor are the same.

If this second check fails, the routine is out of scope and the program is aborted with an appropriate error message.

Otherwise the PROC-word is converted to the "no-check-required" format and stored on the stack.

4.5.3.2 Routine prologue.

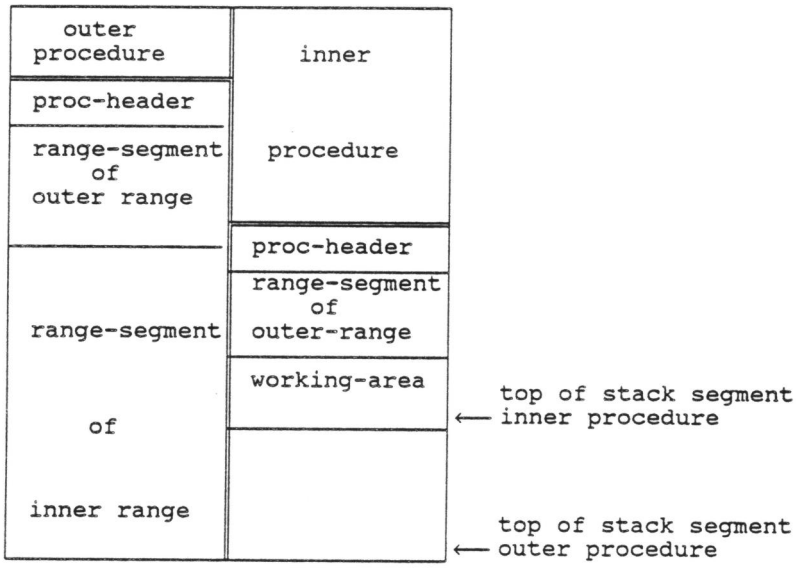
The subroutine G;prol replaces the PROC-word of the callee on the stack by the procedure-header and issues a stack space request for the procedure.

4.5.3.3 Top of allocated stack.

The top of stack is defined as the highest address allocated on the stack. When a procedure is called, its stack segment is allocated above the currently used part of the stack segment of the calling procedure. The stack therefore contains at any point in time for each active procedure only that part of its stack segment that was in actual use at the time the next inner procedure was activated.
The top of allocated stack therefore equals the address of the highest address of the currently used part of some active routine, but not necessarily of the most recently activated routine.

The table below shows a case where the allocated stack is equal to the highest address in use by the outer of two nested routines.

Run time stack requirements of an outer procedure or caller and an inner procedure or callee.



In principle there are two methods to ensure the availability of that longer run-time stack. The first method is to issue a request for space on the run-time stack upon return of the callee. The second method, which we adopted, is never to release the stack space allocated to the caller. The main reason for this choice is efficiency gained by avoiding the overhead on procedure calls implied by the first method.

The top of allocated stack is computed by scanning all existing routine-segments on the run-time stack beginning with the current routine-activation.

For each activation the routine-info entry in the code is located via the routine-header. The allocated length taken from this entry is added to the stack pointer of the activation to determine the top of stack of the activation under inspection.

Then the first instruction at the return-address is fetched. It contains the distance from the caller's to the callee's stack pointer. Then the next outer routine activation is inspected.

The top of allocated stack is computed as the highest top of stack encountered in the scan.

In an earlier version of the system we maintained the top of allocated stack for each routine activation in the procedure header in the run-time stack. The current method is considered more efficient, because it lessens the overhead on assumedly

often occurring procedure calls at the expense of a moderate extra overhead on less frequently occurring garbage collector-activations.

The top of allocated stack is needed in the garbage collector and as an (additional) environment enquiry.

4.5.3.4 Parallel clause.

Execpar is the run-time routine that controls parallel execution. It is called as an ALGOL 68 routine with one parameter of the mode:

```
[ ] STRUCT (PROC VOID x, INT y)
```

Every unit of the par-clause that has never been active, may be activated at any one time. An active unit executing a DOWN operation on a sema must be suspended if the integer referred to by the sema after having been reduced by 1 is not greater than zero.
The suspended unit is associated with the operand sema and may be reactivated when the integer referred to by that sema exceeds zero.

An active unit may itself execute a parallel clause. In that case the unit must be suspended when none of its units can be activated or reactivated. It may be reactivated as soon as at least one of its units may be reactivated.

Applied occurrences of the operator DOWN are translated into calls of the DOWN-routine, which is included in the Execpar module. An applied occurrence of the operator UP causes the integer referred to by the operand sema to be increased by 1.

Execpar performs the necessary initialization, which includes marking its routine-header in the stack (the outermost routine-header is marked the same way). It then searches its units for a (re)startable unit in a manner described below. Effectively it calls the routine of the first unit. When this routine terminates and this was the only unit (left) of this par-clause, Execpar returns, otherwise the entry for this unit is removed from the parameter multiple and the search for a restartable unit is resumed.

When a DOWN operation is executed, the DOWN-routine is activated. In the DOWN-routine the INT value referred to by the operand sema is reduced by one. If the INT value exceeds zero, the DOWN operation is completed.
In the other case the stack pointer of the innermost active par-clause i.e. the innermost activation of Execpar, is located. This is done by inspection of the headers of the surrounding routines for the flag with which Execpar marks its routine-header in the stack.
If the marked routine-header found is the outermost header in the

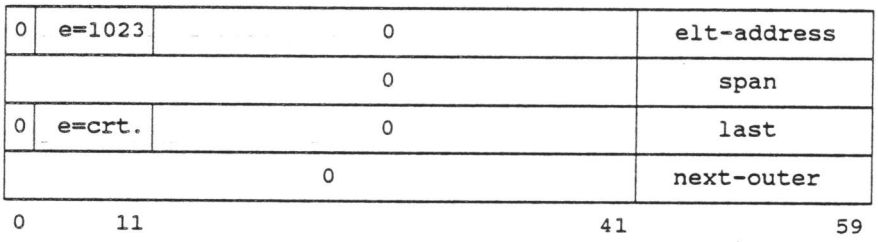
stack, an invalid use of the DOWN operator is diagnosed and the program is aborted.
The stack area associated with the unit to be suspended is copied to the heap.

The units of the par-clause are checked in a round robin fashion and the first unit found to be restartable is given control.
If a unit being checked is itself a suspended par-clause, its units are checked.
At any time the last-field of the Execpar parameter reflects the unit under inspection.

When no unit of the current par-clause is (re)startable, the surrounding par-clause is examined. If no surrounding par-clause exists a deadlock is diagnosed and the program is aborted.

As during this search for a (re)startable unit the stack area of a par-clause being searched is not copied from heap to stack, we may have to perform multiple swaps of stack areas from the heap to the stack before giving control to a restartable unit.
We check whether the par-clause containing the unit to be restarted has its stack area still in the heap. If so we check the next surrounding par-clause until we find a par-clause that has its stack area in the stack.
Then we successively swap the stack areas of the restartable par-clauses from the heap into the stack from outer to inner.
Finally the stack area of the restartable unit found is swapped in and control is given to it, by returning from the activation of the DOWN-routine that caused its suspension, if this was a suspended unit and by calling its routine, if had never been active.

Upon entry of Execpar the dim-field of the parameter multiple descriptor is set to zero, thereby freeing the last two words of the four-word parameter. The parameter area is used as follows:



The first two words represent the descriptor of a zero-dimensional multiple, in which

Elt-address is the address of the multiple of unit-descriptors,

Crt. or current is the index of the unit at which the circular scan of the units started and

Last is the index of the unit most recently having been active. word 3 contains the stack pointer of the immediately surrounding parallel clause.

Once a unit has been activated, it has a stack area associated with it, which begins immediately above the parameters of the surrounding par-clause and ends with the parameters of the routine that triggers its suspension. Thus the last word of such a stack area contains for a unit the sema that is the operand of the DOWN operation and for a par-clause the stack pointer of the surrounding par-clause.

The two word elements of the parameter multiple are associated with the units in the par-clause. Their contents depend on the status of the unit, which may be one of : initial, suspended or active.

state	initial	suspended	active
word 0	descriptor	zero-dimension descriptor-header	garbage collection count
word 1	0	address of copy of stack segment	length of previous heap area

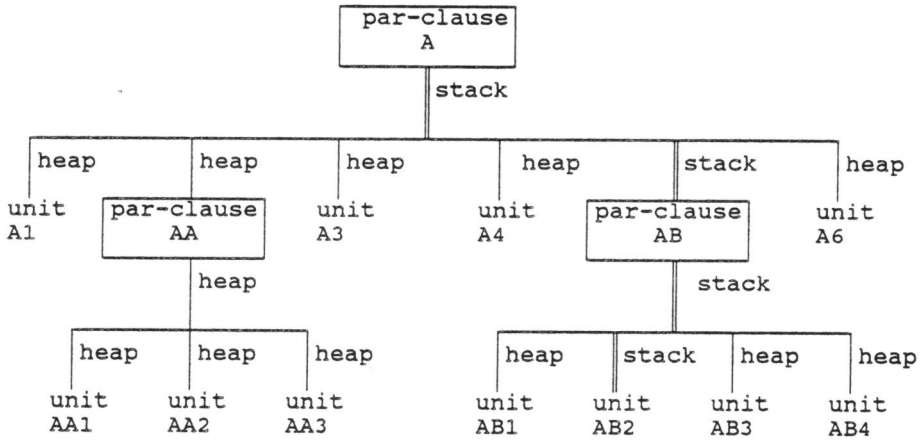
The status of each unit is maintained as follows:
The index of the active unit of an active par-clause is kept in the last-field of the parameter associated with the Execpar activation that initiated the par-clause.
A suspended unit is represented by a multiple descriptor, which can be distinguished by a single instruction from any word i.c. a routine-descriptor, not representing a pointer.

If a unit is activated, its stack area must be swapped in from the heap. The address and length of the heap area are kept in the descriptor of the unit (or par-clause) together with the current of the garbage collection count.
When this unit is suspended this garbage collection count is used to check, whether the garbage collector has been invoked since the unit was swapped in. If the garbage collector has not been invoked the heap area is still available and may be reused, if at least the area to be swapped out is not longer than the area swapped in when the unit was activated.

The stack area associated with a unit or a par-clause begins immediately above the parameters of the surrounding par-clause and ends with the parameters of the routine that triggers its suspension i.e. the DOWN routine for a unit and the EXECPAR routine for a par-clause. Thus the last word of such a stack area contains for a unit the sema that is the operand of the DOWN operation and for a par-clause the stack pointer of the

surrounding par-clause.

As an example, for a program containing a par-clause, say A, of which two units contain a par-clause, say AA and AB, the following diagram shows the par-clauses and their stack areas. The stack areas are indicated by vertical lines. The words "stack" and "heap" at the right of these lines show the current residence of these areas.
The unit labeled "AB2" is the innermost active unit. It is a constituent unit of the par-clause labeled AB, which is initiated from the fifth unit of the par-clause labeled A.



Upon a DOWN operation on sema s1 the sema s1 is checked. If the INT value referred to is zero, unit AB2 is suspended and a scan over the units of par-clause AB is performed. Now if none of units AB3, AB4 and AB1 is restartable, AB is suspended and a scan of the units of par-clause A is performed starting with unit A6. If unit A6 and unit A1 are not restartable, the units of par-clause AA are scanned, starting with the one that had been activated most recently.

Since AA is not active, its stack area resides on the heap. However its units can be scanned without swapping the stack area to the stack. If e.g. unit AA2 is found to be restartable, we must activate first par-clause AA, which means swapping its swap area to the run-time stack, and then unit AA2, which means swapping its stack area back into the run-time stack and passing control to it, by exiting the activation of the DOWN-routine that triggered its suspension.

If however none of the units of par-clause AA is found restartable, the scan of the units of par-clause is resumed with unit A3. If A3 and A4 are not restartable, par-clause A is not restartable and since there is no surrounding par-clause a deadlock is diagnosed.

4.5.3.5 Transput.

Transput has been implemented mainly in ALGOL 68 using the text provided in chapter 10 of RA68. The declarations required in more than one routine are incorporated in a special system circumlude called icftra. System circumludes are described in section 2.2.1.

The ALGOL 68 text in RA68 accurately describes the intention with little regard for implementability. The primitives defined by pseudo-comments in RA68 were implemented using ICF-macros and subroutines written in COMPASS.The actual input/output is done through the Record Manager [50]. Some of the problems encountered are described in the next section.

Ehlich [31] states : "One cannot copy the routines from the report; so one has to write a completely new transput system; so one is forced to understand what is defined by the Report, or, more precisely, what is i n t e n t e d there.".
We found the first two statements to be false and the third to be true. We did copy the routines from RA68 and indeed understanding the intention was required for the implementation of the primitives.

After lengthy discussions in the Task Force on Transput, instituted by IFIP WG 2.1. a model for the implementation of ALGOL 68 transput was established in 1979 by J.C van Vliet in his thesis entitled "ALGOL 68 transput" [30].

Van Vliet used this implementation to test his model. He describes the use of "Record Manager" as interface to the operating system both in his model and in this implementation and the primitives involving real arithmetic for the conversion of numbers to sequences of characters and vice versa, which he retained from this implementation.

4.5.3.6 Interrelating the stack structure and assembly code.

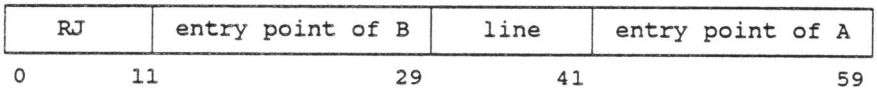
In this section some problems in the incorporation of COMPASS and FORTRAN routines in the ALGOL 68 run-time system will be described.

First we describe some conventions used in FORTRAN and COMPASS code and then we come to the first integration problem.

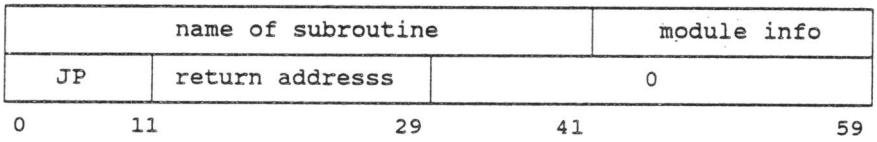
The code generated by the FORTRAN compiler for a subroutine call uses the hardware "RJ" or subroutine jump instruction. This instruction has a single operand denoting the entry point of the subroutine to be called, stores a branch instruction to the address following the RJ-instruction in the entry point word of the subroutine and transfers control to the word immediately following the entry point. Return from the subroutine is usually effected by a branch to the entry point of the callee.

By convention the format of the word in module A containing a

subroutine jump to subroutine B has the format:



RJ stands for the instruction code and line is the source line number in module of the call to B.
Also by convention the word containing the entry point of a subroutine contains the name of the subroutine and the starting address of the module containing the subroutine. The first word of the module contains the name of the module.
The format of the first two words of a subroutine is shown below.



We use the same conventions for COMPASS subroutines.
A call to a COMPASS routine in an ALGOL 68 routine however shows a different format, because either the RJ-instruction is not in the leftmost position of the word or the line-field is zero.

Obviously these conventions make it easy to perform a trace back from most recently activated routine outwards through a number of such routines. Since the code produced for a call from an ALGOL 68 routine is distuinguishable from the code for a call from FORTRAN or COMPASS it is also possible to terminate the trace back when an ALGOL 68 routine is reached.

The above describes some conventions. Next we show the first problem involving the calling of an event routine from transput primitives.

In the first place the COMPASS routines for the transput primitives are not written to be recursive. If however a condition necessitating a call of an event routine is detected from within such a routine that event routine, which may be any ALGOL 68 routine, must be activated. The elaboration of the event routine may then lead to a call of some of the currently active primitive routines and eventually the execution of the calling primitive may have to be resumed.

The solution adopted is as follows:

1. before the event routine is activated a trace back through the currently active COMPASS routines is performed. For every COMPASS routine encountered the address of the entry point and the return address are stored on the run-time stack.

2. the event routine is activated.

3. the return addresses are restored in the entry point words of the COMPASS routines concerned.

If the event routine does not return control to its point of activation step 3 is not executed.

This method is generally usable for the integration of non-recursive routine into the run-time system of a block structured language. The necessary conditions are:

1. the non-recursive routines comply with some convention allowing trace back in reverse order of activation.

2. a call of a non-recursive routine from a module written in the block structured language is distinguishable from a call originating in a non-recursive routine.

Another problem is the general trace back through all different kinds of routines that may be active at any one time.

This implementation supports: ALGOL 68 routines, COMPASS run-time routines, mode dependent FORTRAN interface routines, mode independent FORTRAN interface routines and FORTRAN routines.

A trace back through routines of all these kinds is needed in the top of stack routine described in section 4.5.3.3 as well as in the diagnostic system and is made possible by generating different instruction sequences for calls of routines of different types.

4.6 Garbage Collector.

The task of the garbage collector is to make the memory occupied by data in the heap that is no longer accessible, available to the program. Garbage collection for ALGOL 68 is described by Wodon [26] and Branquart [20].

The distinctive feature of the garbage collector in this implementation is, that it requires only the data in the stack and the heap.

The internal representation of a pointer in the ALGOL 68 system contains all the information necessary for the marking of the pointers. The addresses of data items accessible to active FORTRAN routines are found by means of parameter-list descriptors in the run- time stack.

The garbage collector algorithm logically consists of four tasks: marking, sorting, pointer adjustment and data compaction.

The marking task comprises locating and chaining all existing pointers and to determine the lowest address of all data items accessible to an active FORTRAN routine.

As in the Cyber ALGOL 68 system pointers contain not only the address but also the size of their objects these pointer chains contain all information required for the pointer-adjustment and data-compaction tasks.

The sorting task rechains the marked pointers in the order of descending address of their object in preparation of the pointer-adjustment task.

The adjustment task then in a single scan through the ordered chains of pointers locates all objects to be retained in the order of descending original address, determines a new address for all objects and adjusts the pointers accordingly. In this scan also a chain of unused memory segments or holes is established.

The compacting task performs a single scan of the chain of holes.

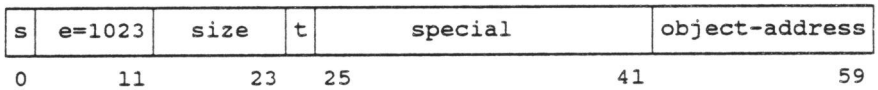
In this scan all accessible data below the data accessed by some FORTRAN routine is moved upward into one compact memory segment.

4.6.1 Marking phase.

An object is accessible when it resides in the stack segment of an active block in the program or when it is the object of an accessible pointer. Since objects may contain pointers the set of accessible objects is recursively defined.

Pointers are distinguishable from all other values. A pointer, when regarded as a floating point number, represents an infinite

value. A pointer and an infinite value generated by a hardware instruction differ in that the mantissa of a pointer is non-zero and the mantissa of an infinite value is forced to zero by the hardware. The format of the first word of a pointer value is:



where the meaning of the fields is as follows:

s discriminates references and descriptors:

s=0 for a value of mode REF amode, also called a reference. Size is the number of words occupied by the object of the pointer minus one,

s=1 for a value of mode ROWS amode, also called a multiple-descriptor. Size is the number of words occupied by an element of the multiple minus one,

t signals the presence of pointers in the object:

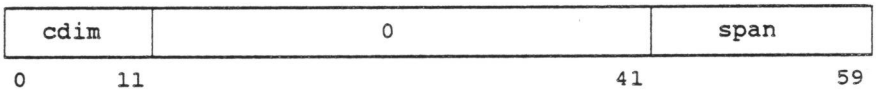
t=0 if the object is plain i.e. does not contain pointers,

t=1 if the object may contain pointers.

special contains several subfields of which we mention here only the dim-field denoting the number of dimensions of the multiple.

A multiple descriptor occupies 2+2*dim words, where dim >= 0.

The second word of a descriptor has the format:



where

cdim is a metric for the compactness of the multiple defined as the minimum number of nested loops required to access all the elements of the multiple,

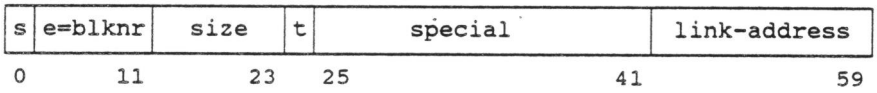
span is the length in words of the smallest memory segment containing all elements of the multiple.

The remaining 2*dim words represent the bound-information. Each of the dim so-called triples consists of two words, of which the first contains the lower bound and the second contains the width and the stride in one dimension of the multiple.

Width is the number of elements i.e. upper bound-lower bound+1 and stride is the distance in words of two elements of the multiple of which the subscript in the dimension considered differ by 1.

Thus the length of the object is found in a reference as one more than the value the size field and in a descriptor as the value of the span field.

A word is recognized as a pointer by the value of the exponent field. When a pointer is found, it is converted into the format:



The object-address and the length of the object minus one are added to yield the address of the last word of the object. The result is then split into two fields: the most significant 9 bits, called block number or blknr, and the lower 8 bits, called the offset. The word containing the pointer is chained, using its address-field to hold a link. The block number is stored into the exponent field.

As will be discussed later, we need the address of the last word of the object rather than of its first word, because data is compacted in an upward (to higher addresses) direction.

The reformatted pointer is no longer recognizable as a pointer to be marked because its exponent-field now represents an exponent value less than 512, rather than equal to 1023 as in the original form of the pointer.

In order to be able to reconstruct the pointer in a later phase of garbage collection (the adjustment phase) we need to remember the offset computed. This is achieved by maintaining a chain for each of the 256 possible values of the offset and to chain each pointer in the chain corresponding with the offset of its objects last word address in its memory block.

We establish two sets of 256 chains of pointers of which one is called the set of *ready-chains* and the other the set of *follow-chains*. Each chain of pointers requires a single header word to hold the address of the first pointer in the chain. New entries are always appended at the head of the chain.

Ready-chains contain pointers, of which the objects are known not to contain non-marked pointers and follow-chains contain pointers of which the objects may contain not yet marked pointers.

The object of a pointer is known not to require marking, if the mark-flag or t-bit of the pointer is zero or if the pointers contained in its object have been marked. Thus a pointer with

t-bit = 0 is transformed as described above and chained in the appropriate ready-chain.

A pointer with t-bit = 1 is also transformed, but further processing may be done in two different ways: it may either be chained in the appropriate follow-chain, to be further processed at a later time or alternatively all pointers contained in its object may be processed immediately and the pointer itself chained in the appropriate ready-chain. The choice between these two methods is a matter of efficiency only and in general the decision is made on the basis of availability of registers.

4.6.1.1 Scanning a compact object.

There is a subroutine called FOLREF, which marks all pointers in a compact segment of memory. FOLREF is designed to mark the objects of the pointers it recognizes as long as no additional registers are needed.

FOLREF maintains the address of the word currently being examined and the end address of the segment to be examined in two hardware registers. It performs a linear scan of the segment.

During the scan the current is examined as follows:

1. if the word is plain, no action is needed.

2. if the word is a pointer, the t-bit is examined:

   1. if the t-bit is zero, the object of the pointer does not contain any pointer. the pointer is chained in the appropriate ready-chain and the processing of the pointer is completed.

   2. if the t-bit is one, the length of the object is examined in the following steps:

      1. if the length is zero words (an empty multiple) the pointer goes into a ready-chain and no further action is needed.

      2. if the object is one word long, the pointer is chained in the appropriate ready-chain and the object is examined using the same registers as used for the examination of the original pointer.

      3. if the object is more than one word long, we check:

         1. compactness of the object (this check fails only for multiple descriptors with cdim greater than zero)

         2. equality of the last word addresses of the pointer (which may be a descriptor and several words long) and the segment to be examined.

If one of the checks fails the pointer is chained in the appropriate follow-chain, otherwise the pointer goes in a ready-chain and FOLREF is executed for the object of the pointer.

Note, that pointers with t-bit = 0 are not checked for compactness of their object i.e. no attempt is made to recover the holes in a slice, because we need to leave the indexing of the slice unchanged.

The problem of the marking of the holes in a slice has been discussed by Branquart [20], who calls them "interjacent spaces between the static parts of the elements of a multiple".

In the solution he describes the internal representation of a subvalue of a given multiple contains a pointer to the descriptor of that given multiple and the latter descriptor is used to mark the interjacent spaces This method is unduly complicated since the descriptor of the subvalue contains all the information required.

However the inclusion in the descriptor of a subvalue of a multiple to the descriptor of that multiple could be put to use. If during the marking phase all descriptors of subvalues of a given multiple were placed in a chain from the descriptor of that multiple, it would become possible to try to drop some or all interjacent spaces and to adjust the strides in the descriptors of subvalues of the given multiple accordingly.
In Branquart's example the given multiple, identified by A, is of mode "[1:3,1:2] REF μ" and the only subvalue considered is equal to A [,1] and hence of mode "[1:3] REF μ". The stride in the descriptor of the subvalue would be equal to the stride in the second quintuple (original Report!) of the descriptor of the given multiple and would be equal to two in units of storage used for a value of mode REF μ.
If no references into the interjacent spaces (A[1,2] and A[2,2]) exist, dropping the interjacent spaces would be possible provided the stride in the descriptor of the subvalue were changed to one in the same units as before.

4.6.1.2 Stack scan.

The marking phase first scans the run-time stack and chains all pointers it recognizes. As in FOLREF an attempt is made to minimize the number of pointers in follow-chains.

Pointers with t-bit = 0 point to plain objects and are chained into the ready-chain corresponding to the offset of their objects last word address.

Pointers with t-bit = 1 are examined in more detail. The length of the object is considered.

if the length is zero, the pointer goes in the appropriate

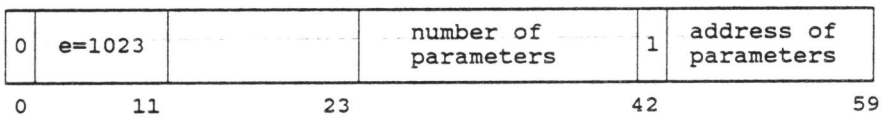
ready-chain.

if the length is one, the pointer is chained in the appropriate ready-chain and the object is examined in lieu of the pointer.

if the length exceeds one and the object is compact, the pointer goes in a ready-chain and FOLREF is called to process the object.

Note, that the pointer to an object must always be transformed and chained before the object is examined, because circular references might otherwise cause the garbage collector to go into an endless loop.

In the stack scan also FORTRAN parameter-list descriptors are recognized. We show once more their format:



A parameter-list descriptor is distinguished from a multiple descriptor by the 1-bit in bit position 42.

The stack scan maintains a variable called GARLIM, which contains the lowest address of any data item accessible to an active FORTRAN routine.
For every descriptor found the addresses of all parameters in the list are compared to the address contained in GARLIM. If the parameter address is lower, it is stored in GARLIM.

GARLIM will be used in the adjustment phase to enforce the rule, that data accessible to a FORTRAN routine must not be moved.

4.6.1.3 Completing the marking.

The second step is an iterative process. In each iteration all follow-chains are scanned. The objects of the pointers in the chain are inspected for pointers. When the scan of a follow-chain has been completed, it is appended to the corresponding ready-chain.
The process is terminated when all follow-chains are empty.

The pointers in the follow-chain being scanned are processed as follows:

The last word address of its object is reconstructed from the block number stored in its exponent field and the offset particular to the follow-chain being scanned.

The t-bit of the pointer is not examined, because follow-chains contain only pointers with t-bit = 1 i.e. pointers to objects

that may contain pointers. Each word of the object is checked for an infinite exponent. When the check succeeds a not previously marked pointer is recognized.

If the object of the pointer is a plain value the pointer is chained in the ready-chain corresponding to the offset of the last word address in its block.

Otherwise the pointer is processed as follows: the pointer is chained in the appropriate ready-chain. Further processing depends on length and compactness of the pointer's object.

A zero length object is completed .

A one word object is examined in lieu of the pointer itself.

A compact object exceeding one word in length is passed to FOLREF.

Finally a non-compact object is broken in its constituent compact parts. Each part is processed by a call to FOLREF. The number of nested loops needed to find all parts of the slice is found in the cdim field of the descriptor.

The scanning of a particular follow-chain may cause newly marked pointers to be linked into the same follow-chain. Such pointers are not processed in the same scan, because pointers are linked directly after the head of the chain rather than at its tail.

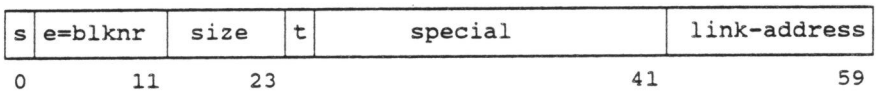
Before the scan of a follow-chain its head is copied in a local variable and the head is cleared. Then the chain is scanned. After the scan the chain is linked in front of the corresponding ready-chain.

In each iteration all follow-chains are scanned. Before each scan a flag is reset to indicate no pointer has been linked in a new follow-chain. When a pointer is linked into a follow-chain during the the iteration, the flag is set. At the end of the iteration the flag is tested. If it is set a new iteration is started, otherwise the process is terminated.

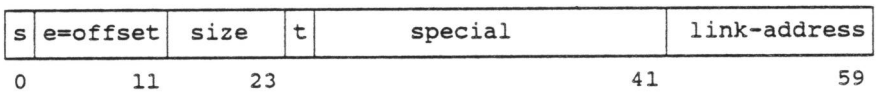
When the iterative process is terminated all accessible pointers have been chained in the ready-chain corresponding to the offset of the last word of their object.

4.6.2 Sorting phase.

For each block in memory a *new-chain* is established. The sorting phase chains all pointers found by the marking phase into these new-chains. Each chain pointers are linked in the order of the offset of the last word address of their object. In this process the pointers are transformed from the format:



into the format:



the last word address of a pointers object is now partly in the identity of the chain containing the pointer and partly in the e-field of each pointer.

The ready-chains created by the marking phase are processed in order of increasing offset. Every pointer in each chain is rechained in the new-chain corresponding to the block containing the last word of their object. Since pointers are always inserted at the head of a chain and rechaining is done in order of increasing offset the pointers in each new-chain are linked in order of decreasing offset.

All accessible pointers may thus be accessed in decreasing order of address of the last word of their objects by scanning the new-chains in decreasing order of block number.

The sorting phase is similar to the second pass of a two column bucket sort well known from the hollerith card sorters.

4.6.3 Pointer adjustment phase.

The compacting phase will move all accessible objects in the heap that are located below the address in GARLIM into a contiguous area of memory extending downward from that same address. The ordering of objects in memory will not be changed.
The move proceeds from the highest to the lowest source address. Every word is moved upward in memory over a distance equal to the number of words found unused between address contained in GARLIM and its original address.

The pointer-adjustment phase performs a single scan of all accessible pointers in order of decreasing last word address of their object. This scan is split in two parts.

The first part processes pointers to objects in the heap. It maintains an offset and the lowest address of accessible objects located so far as well as a list of segments of unused words or holes. The value of the offset is kept equal to the number of words in all the holes located below GARLIM that have been found so far.

For each pointer scanned a check is made for unused words between

the last word of the object and the current lowest address. If such a hole is found it is appended to the list of holes and if the hole is located below GARLIM, the current offset is adjusted. The pointer is reconstructed in its original form and the current offset is added to its address field.

The second part processes pointers to objects in or below the run-time stack and restructures the pointers into their original form.

4.6.4 Compacting phase.

This phase uses the information in the list of holes to move all data below GARLIM and between holes upward into a compact area extending downward from GARLIM.

The area from the top of allocated stack to the heap is designated for extension of the heap and is padded with SKIP values. This padding is performed here because padding a large area at a time is more efficient than padding a small area on each heap request.

4.6.5 Garbage collector memory requirements.

The memory requirements of the garbage collector are very modest:

headers of chains of pointers:	
ready-chains	256
follow- and new-chains: max(256,512) =	512
local variables	24
code	308
total	1100 words

4.6.6 Interface between program and garbage collector.

The garbage collector may be invoked as the result of a heap request, a stack request or a direct garbage collector invocation. The garbage collector provides some additional environment enquiries.

4.6.6.1 Heap request.

A heap request for a given number of words on the heap returns a pointer to the new area. Upon return the new area has been initialized to SKIP values.

A heap request directly from an executing ALGOL 68 program is processed as follows:

1. An attempt is made to satisfy the request from the area designated for heap extension. The beginning- and end-addresses of this area are in the dedicated registers B4 and B2. Since the designated area is padded with SKIP values, no further action is required.

2. The list of holes is searched for a hole large enough to satisfy the request. The area granted is padded with SKIP values.

3. The garbage collector is invoked.

4.6.6.2 Stack request.

A request for extension of the run-time stack is processed in the following steps:

1. If the extended stack remains below the designated heap extension area the request is completed.

2. If the extended stack remains below the heap, the dedicated register is set at the first free word above the stack and the request is completed.

3. The garbage collector is invoked.
   A check is made, whether the extended stack now fits below the heap:

   If the check succeeds, the request is completed.Otherwise more memory is requested from the operating system and the garbage collector is activated once more.

   If after the second garbage collection the request cannot be satisfied the program is aborted with an appropriate message.

4.6.6.3. Additional environment enquiries.

The additional environment enquiries related to the garbage collector include:

Size of stack.
Size of allocated stack.
Size of the heap.
Garbage collector activation count.
Garbage collector cumulative Central Processor time.

5 CONCLUSIONS.

5.1 Characteristics of the Cyber ALGOL 68 Implementation.

An almost complete implementation of ALGOL 68 Revised has been realized. The extensions to the language include removal of most restrictions imposed by the ALGOL 68 scope rule, separate compilation facilities, a method to allow the definition of operators in terms of machine instructions, called ICF-macros, and a FORTRAN interface.

Preludes and postludes may be nested, but each level of nesting requires compilation of the innermost level prelude and postlude.

The treatment of the scope rule restricts run-time scope checks to the checking of the scope of a routine at the time the routine is activated.

The compiler executes six complete scans of the source program in some intermediate form. The first three scans are largely machine independent and perform the lexical, syntactic and semantic analysis of the source text. The remaining scans generate, optimize and edit the object code.

The run-time system employs a stack and a heap to store data. The stack contains fixed size segments for each active procedure and all data that cannot be allocated in fixed segments or that is referred to by pointers, of which the scope is not known at compile time, is allocated on the heap.

This approach simplifies stack handling at the expense of heap activity. Therefore a simple and fast garbage collector is required. Pointers are internally represented as out of range floating point numbers in a format acceptable to but not generated by the floating point hardware instructions.

This representation of pointers simplifies the recognition of pointers in the marking phase of the garbage collector and also makes it possible to allocate the links of marked pointer chains in the pointers themselves.

The garbage collector based on this representation is simple and fast and does not require any special provisions in the generated code other than initialization of all data to a bit pattern that is distuinguishable from the patterns used to represent pointers.

By exploiting the hardware characteristics of the Control Data computer system we succeeded in building a fast and elegant garbage collector without placing a burden on the compiler.

5.2 Applicability of Some Concepts in Other Environments.

Some ideas implemented in this compiler may be generalized and used in other environments.

The use of the ALGOL 68 "library-prelude" concept in combination with ICF-macros and a separate compilation facility to implement the standard prelude could fruitfully be applied in the implementation of compilers for almost every programming language.

To recall, ICF-macros are used in this implementation to define operators in terms of ICF-instructions, i.e. in the language in which the file generated by the code generation phase is written.

In fact in the implementation of an ALGOL 60 compiler [43] for the Control Data Cyber 170 series of computers these techniques have been applied. In that implementation the use of ICF-macros has been extended to the definition of procedures.

One of the motives behind the development of ICF-macros was to separate the implementation of the many operators in the standard prelude from the implementation of the code-generator phase of the compiler, in order to ease the planning of the project.

At the same time definition of standard operators was greatly simplified and maintenance (an implementers euphemism for correction of errors after release of the product) even more so. Correction of an error in an operator definition is effected by correcting the ICF-macro in the source text of the standard prelude, compiling the standard prelude and replacing the appropriate compiler overlay with the output of the compilation.

An interesting example of the use of ICF-macros in a special purpose circumlude is the implementation by J.C. van Vliet of the primitives required for his transput model described in his thesis [30].

Another example is the implementation of interval arithmetic described by Günther and Marquardt [32].

A following step would be the use of ICF-macros to define code sequences for particular language constructs.
With the existing limited syntax of ICF-macros this usage would be restricted to simple constructs like coercions, subscripting of one dimensional multiples and selections not involving multiples.

Extending the syntax of ICF-macros and definition of additional compile-time variables would be needed to allow definition of the more complex code sequences for slicing, assignment ,declarations etc.

Following such an approach would ultimately lead to compilers in which a substantial part of the code sequences are defined in the

source text of a circumlude rather than in the code generation phase of the compiler.

Although the garbage collector in this implementation depends heavily on the characteristics of the target machine, this dependency applies mainly to the tracing part of the marking phase.

In a system in which we cannot dedicate some bit patterns to the representations of pointers only, the compiler would have to provide mode-information concerning the data in the run-time stack and another algorithm identifying pointers in the marking phase would be required.

The method of marking pointers by transformation and the method of chaining marked pointers however remain applicable provided the internal representation of pointers have room for an index into the mode-information table, a link to chain the pointer in a chain of pointers, a mark bit and half an address for the object of the pointer. The other half of the address of the pointer would be kept in the identity of the chain containing the pointer.

The design of the sorting, pointer-adjustment and compaction-phases contain no further hardware dependencies.

5.3 Cyber ALGOL 68 and Compiler Designs in Literature.

In this section we compare the design of this compiler to some other designs found in literature.

5.3.1 The Berlin ALGOL 68 implementation.

Koch and Oeters present an outline of the Berlin implementation in [33]. Some characteritics are (partial) portability, separate compilation facilities and the need for a dedicated linker.

In this implementation no dedicated linker was deemed necessary because of the mode checking during separate compilation described in section 2.5.3, which ensures that the declaration of a procedure as external and the separate compilation of a procedure associate a given external name with one and the same mode.

As portability is not a design goal for this implementation, we will limit the comparison to the machine independent parts of the two compilers.
Comparing the compiler structures then, we observe that the machine independent part of the Berlin compiler consists of four passes, whereas this implementation contains only three machine independent passes.

The first passes of both compilers are roughly equivalent.

The second pass of this implementation performs a mode independent parse of the program like the second pass of the Berlin compiler and additionally builds an initial mode table.

The third pass of the Berlin compiler builds the mode table, extends it for modes resulting from coercions, selections and slicing and performs mode equivalencing.

The third pass in this implementation performs mode equivalencing and handles identification and coercions, extending the mode table for modes resulting from coercions, selections and coercions in the process, while in the Berlin compiler identification and coercions require a fourth pass.

The key factor in our three pass design is the capability of extending the mode tables with the modes resulting from coercions, selections and slicing after the mode equivalencing, which implies extending the mode table avoiding the generation of multiple entries in the mode table representing the same mode. The technique employed to extend the mode table preserving the unicity condition is described in section 3.6.4.2.4.

The ALGOL 68 R 4000 Compiler, described by Loeper, Jäkel and Pietsch [35] shows a four pass machine independent, in which the functions of the four passes are similar to those of the Berlin compiler.

5.3.2 The Paris ALGOL 68 compiler.

A brief sketch of a compiler implemented by the University of Paris IX (Orsay) on Univac 1100 is given by Taupin [34].

This is an eight pass compiler. The first four passes form the machine independent part.

The first pass performs roughly the same functions as the first or lexical phase in the Cyber implementation. The output file of this pass is a random access file and therefore information pertaining to a block that becomes available when the end of the block is reached can be stored at the start of the block on the output file, while in the Cyber compiler such information is stored in a symbol table entry created when the start of the block is encountered, which appears to be more efficient.

The second pass, similar in function to the second phase of the Cyber compiler is unusual in that it does not create an output file, but stores all information gathered in memory resident tables.

The third pass, reading the output of the first pass and using the tables generated by the second pass, performs both semantic processing, producing coercions, and code generation. Mode equivalencing is done of the fly and, maybe therefore, is

incomplete. The code generated is not object machine language, but an internal three-operand code. Coercion entries, which are often determined after the coercend is output, are stored with the coercend by means of random access of the output file. This pass is functionally similar to the third and fourth passes of this implementation, except that it produces a code in which coercions can be inserted, which is of course not possible for machine code. The incompleteness of the mode equivalencing would be unacceptable in this implementation.

The fourth pass is an optimizer pass. It recognizes common subexpressions and deletes unused code. These functions are done in the mode tree building routine of the (fifth) code optimizer phase in the Cyber implementation. The latter is preferable since these optimizations should be executed on atomic operations to be optimally effective.

5.4 Cyber ALGOL 68 and Run Time System Designs in the Literature.

5.4.1 The ALGOL 68C compiler.

Birrell [37] gives a clear description of the storage management used in the ALGOL 68C compiler. In the sequel I point out some similarities and some differences.

The multiple descriptors he describes are quite similar to those in this implementation. He proposes to store in the descriptor the (virtual) address of the element with subscript 0 in every dimension instead of the address of the first element in order to simplify subscripting.
This is interesting idea, but it has its problems too. The difference of the two addresses may lie outside the range of integer values supported and therefore in in its computation and in subscripting one needs multiple precision arithmetic or at least arithmetic modulo some convenient number greater than the highest address value allowed.
In a two's complement machine this causes little overhead given a suitable instruction repertoire, but in one's complement mode the overhead involved incurred could easily exceed the savings, at least for multiples with more than one dimension.

For objects of mode REF [...] BMODE Birrell describes an optimization, which may be traced back to a discussion between Branquart and Mailloux found on page 235 in [20], and that consists of representing such a value by a descriptor of a multiple instead of by a pointer and the descriptor it points to. He remarks that this optimization causes the implementer some tedium, since REF [...] AMODE must always be handled as a special case. Since in our project timely completion had a higher priority than optimization, we did not implement this optimization.

Birrell discuses three schemes for the run-time stack, which

differ in the allocation dynamically sized objects of non-global scope only.

In all three schemes the stack segment for a procedure contains a number of fixed length subsegments for the fixed size data associated with the ranges of the procedure. These subsegments are allocated in the order of nesting of the pertinent ranges.

In the first scheme the dynamically sized data associated with the nested ranges are allocated in the stack segment for the procedure above those fixed length subsegments.

In the second scheme the dynamically sized data are allocated in a second stack.

In the third scheme all dynamically sized data is allocated on the heap.

In the first and second scheme a pointer to the top of the dynamically sized data for a range is maintained within the subsegment containing the fixed size data for that range.

In the first scheme the allocation of the fixed and dynamic parts of parameters poses great problems: if the fixed parts are to be treated as the outermost range of the callee, these fixed parts must be allocated at the top of the calling procedure's stack segment and then the dynamic parts can not be allocated in that stack segment. Birrell therefore rejects the first scheme.

He rejects the third scheme, because he considers it expensive and because it uses the heap behind the programmers back.

In ALGOL 68C therefore the second scheme is implemented. Two stacks are maintained: a "static" stack, similar in function and layout to the stack in the Cyber implementation, and a "dynamic" stack holding all data structures with dynamically computed size of non-global scope.

In this implementation the third scheme is adopted. In view of the design of the garbage collector we did not consider this scheme too expensive and we have no objection to using the heap "behind the programmers back".

ALGOL 68C and this implementation both use a chain of static links to provide access to data in stack segments other than that of the innermost active routine.
The similarity between the rationale given in Birrell's article and the one presented in section 4.4.1. of this thesis is striking. One of the arguments given in both is, that the number of static links in a chain is usually less than the textual nesting level, since if an enclosing stack segment is not referred to from within a procedure it may be omitted from the chain.
Birrell correctly adds : "this is allowed by and required by the rules on scope of routine values". However the argument holds for

other block structured languages as well. In fact in the ALGOL 60 implementation for the Cyber computers a chain of static links à la ALGOL 68 is realized.

5.4.2 The Munich compiler.

In her lecture [38] presented during the Munich "Advanced Course on Compiler Construction" U. Hill bases her discussion on the Munich Compiler.

The stack organization follows the first scheme described in section 5.3.1 above and shows a procedure based allocation of static data and range based allocation of dynamic data.

For assignations and declarations concerning modes showing multiples of which the components show multiples a partly interpretative method is employed. In the Cyber implementation the code for all such constructs is generated at compile time. Rather than interpretation, generalized run-time routines are used. The latter method has the advantage that it is relatively easy to single out special cases that are recognizable at compile time for optimization either by providing specialized and faster run-time routines or by replacing some calls to run-time routines by inline code.

A ghost element is maintained for empty multiples of which the components contain multiples.

In order to allow dynamic scope checking values of mode REF contain two address, one for the object referred to and one for the block that is the scope of the reference.

The garbage collector differs considerably from the Cyber implementation and consists of three phases:

1. Marking phase.
   First all pointers in the static areas of active block and procedure activations are marked. For each static area a so called storage-allocation list, generated by the compiler, is used, which contains for each item in the static area a reference to the appropriate entry in a run-time mode table.
   From the pointers located in the first step pointers in the heap are marked.
   The marking is done by setting bits in a bit map.
   A list of pointers is created in this phase.
2. Compacting phase.
   Compacting is performed using the bit map created in the marking phase.
3. Pointer adjustment.
   This phase is facilitated by the list of pointers created in the marking phase.

For each static area a single storage-allocation list exists and therefore objects of different non-primitive modes (non-plain

modes in our terminology) must not be stored in the same location in a static area. This restriction increases the size of the static area, complicates the data allocation in the compiler and necessitates additional code to clear no longer used intermediate results in the static area.
Also the storage-allocation list for a given block is represented in the stack by a number rather than by its run-time address, causing problems with separate compilation.

As pointer adjustment is done after compacting the adjustment of a pointer requires the new address of the pointer as well as the new address of its object. Performing pointer adjustment before compacting, as in the Cyber implementation, is simpler because one needs to determine only the new address of the object of the pointer.

6 REFERENCES

[1] Report on the Algorithmic Language ALGOL 68, A. van Wijngaarden (Editor) , B.J. Mailloux, J.E.L. Peck and C.H.A. Koster Mathematical Centre MR 101 February 1969.

[2] Revised Report on the Algorithmic Language ALGOL 68, edited by A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens and R.G. Fisker, Springer-Verlag, 1976.

[3] Report on the Algorithmic Language ALGOL 68, C.H.A. Koster, C.H. Lindsey, B.J. Mailloux, J.E.L. Peck, M. Sintzoff, A. van Wijngaarden, August 1972.

[4] Draft Revised Report ALGOL 68, working Document of W.G.2.1, February 1973.

[5] Draft Revised Report on the Algorithmic Language ALGOL 68, Editorial Working Document of W.G.2.1, April 1973.

[6] Proposed Revised Report on the Algorithmic Language ALGOL 68, Edited by A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff and C.H. Lindsey with the assistance of L.G.L.T. Meertens and R.G. Fisker, July 1973.

[7] Almost the Revised Report on The Algorithmic Language ALGOL 68, Edited by A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens and R.G. Fisker, September 1973.

[8] Report on considered improvements ALGOL Bulletin 34.3.2, July 1972.

[9] Proposals for revision of the Transput Section of the Report ALGOL Bulletin 34.3.3, July 1972..

[10] Further Report on Improvements to ALGOL 68 ALGOL Bulletin 35.3.1, March 1973.

[11] Final Report on Improvements to ALGOL 68 ALGOL Bulletin 36.3.1, November 1973.

[12] Language changes Incorporated in the Revised ALGOL 68 Report, B. Uzgalis, July 1973.

[13] Hansen, W.J. and Boom, H., The Report on the Standard Hardware Representation for ALGOL 68, ALGOL Bulletin 40.5, August 1976.

[14] Moudry , J., Kral, J., Nachrchal, J. and Sokol, J., Recognition of Routine Denotations in ALGOL 68, ALGOL Bulletin 35.4.4, March 1973.

[15] De Remer, F.L., Simple LR(k) Grammars. Comm. A.C.M. 14,7 (July 1971)

[16] Koster, C. H. A., On Infinite Modes, ALGOL Bulletin 30.3.3, February 1969.

[17] Zosel, M.E., A formal grammar for the representation of modes and its application to ALGOL 68, Ph.D. Thesis, Computer Science Group, Un. of Washington, 1971.

[18] Goos, G., Some problems in compiling ALGOL68, in: ALGOL 68 Implementation (J.E.L. Peck, ed.), Proceedings of the IFIP Working conference on ALGOL 68 Implementation, Munich, July 20-24, 1970, North-Holland Publ. Cy., 1971.

[19] Branquart, P., Lewi, J. and Cardinael, J.P., Analysis of the Parenthesis Structure of ALGOL 68, ALGOL 68 Implementation (J.E.L. Peck, ed.), Proceedings of the IFIP Working conference on ALGOL 68 Implementation, Munich, July 20-24, 1970, North-Holland Publ. Cy., 1971.

[20] Branquart, P. and Lewi, J., A Scheme of Storage Allocation and Garbage Collection for ALGOL 68, ALGOL 68 Implementation (J.E.L. Peck, ed.), Proceedings of the IFIP Working conference on ALGOL 68 Implementation, Munich, July 20-24, 1970, North-Holland Publ. Cy., 1971.

[21] Mailloux, B.J., On the Implementation of ALGOL 68, Ph.D. Thesis, Mathematisch Centrum, 1968.

[22] Meertens,L., On Static Scope Checking in ALGOL 68, Algol Bulletin 35.4.8, March 1973.

[23] Oostrum, P. van, A Parser for ALGOL 68, Masters Thesis, Utrecht, January, 1977.

[24] Hedel, H.W.M. van, Mode Handling in ALGOL 68, Masters Thesis, Delft, March 1979.

[24] Schlichting, J. J. F. M., De Cyber Algol 68 Vertaler, Colloquium Capita Implementatie van Programmeertalen, MC Syllabus 42, Mathematisch Centrum, Amsterdam 1980.

[25] ALGOL 68 Version II Reference Manual, Control Data B.V., 1984.

[26] Wodon, P. L. Methods of Garbage Collection for ALGOL68, ALGOL 68 Implementation (J.E.L. Peck, ed.), Proceedings of the IFIP Working conference on ALGOL 68 Implementation, Munich, July 20-24, 1970, North-Holland Publ. Cy., 1971.

[27] Meulen, S.G. van der, and Veldhorst, M., Torrix, a Programming System for Operations on Vectors and Matrices over Arbitrary Fields and of Variable Size, Mathematical Centre Tracts 86, Amsterdam, 1978.

[28] Grune, D. Naamlijst-algorithmen, Colloquium Capita Implementatie van Programmeertalen, MC Syllabus 42, Mathematisch Centrum, Amsterdam 1980.

[29] Grune, D. The MC ALGOL 68 Test Set, IW 53/75, Mathematisch Centrum, 1975.

[30] Vliet, J.C. van, ALGOL 68 TRANSPUT, Thesis, Mathematisch Centrum, 1979.

[31] Ehlich, H. and Wupper, H., Experience with ALGOL 68 Transput and its Implementation, Arbeitsberichte des Rechenzentrums der Ruhr-Universität Bochum, ISSN 0341-0358, Nr.7901, 1979.

[32] Günther, G. and Marquardt, G., A Programming System for Interval Arithmetic, Proceedings International Conference on ALGOL 68, Mathematical Centre Tracts 134, Mathematisch Centrum, Amsterdam,1981.

[33] Koch, W. and Oeters C., The Berlin ALGOL 68 Implementation, Proceedings of the Strathclyde ALGOL 68 Conference, SIGPLAN Notices, Volume 12, Number 6, June, 1977, page 102-108.

[34] Tauplin, D., The ALGOL 68 Compiler of Paris - XI University (Orsay), Proceedings of the Strathclyde ALGOL 68 Conference, SIGPLAN Notices, Volume 12, Number 6, June, 1977, page 117-128.

[35] Loeper, H., Jäkel, J. and H.Pietsch, H., Semantic Analysis and Synthesis in the ALGOL 68 R 4000 Compiler, Proceedings International Conference on ALGOL 68, Mathematical Centre Tracts 134, Mathematisch Centrum, Amsterdam,1981.

[36] Penello, Thomas J, Very Fast LR Compiling, Sigplan Notices, Volume 21, Number 7, July 1986.

[37] Birrell, A. D., Storage Management for ALGOL68, Proceedings of the Strathclyde ALGOL 68 Conference, SIGPLAN Notices, Volume 12, Number 6, June, 1977, page 82-94.

[38] Hill, U, Special Run-time Organization Techniques For ALGOL 68, Lecture Notes, Advanced Course on Compiler Construction, Tech. University of Munich, March, 1974.

[39] Waite, W. M. and Goos, G., Compiler Construction, Springer Verlag, New York, NY, 1984.

[40] Aho, A. V., Sethi, R. and Ullman, J. D., Compilers: Principles, Techniques and Tools, Addison-Wesley, Reading, Mass., 1986.

[41] Gries, D., Compiler Construction for Digital Computers, John Wiley, New York, 1971.

[42] Cleveland, J. C. and Uzgalis, R., Grammars for Programming Languages, Elsevier, New York, 1977.

[43] ALGOL 60 Version 5 Reference Manual, Control Data Corporation, Publication No. 60481600.

[44] NOS Version 1 Reference Manual Volume 1, Control Data Corporation, Publication No. 60435400.

[45] NOS Version 1 Reference Manual Volume 2, Control Data Corporation, Publication No. 60459670.

[46] NOS/BE Version 1 Reference Manual, Control Data Corporation, Publication No. 60493800.

[47] SCOPE Version 2 Reference Manual, Control Data Corporation, Publication No. 60342600.

[48] SYMPL Version 1 Reference Manual, Control Data Corporation, Publication No. 60496400.

[49] Compass Version 1 Reference Manual, Control Data Corporation, Publication No. 60492600.

[50] Cyber Record Manager Version 1 Users Guide, Control Data Corporation, Publication No. 60359600.

[51] FORTRAN Version 5Reference Manual, Control Data Corporation, Publication No. 60481300.

[52] CDC Cyber 170 Computer systems Models 720, 730, 740, 750 and 760, Model 176 (level B), Control Data Corporation, Publication No. 60456100.

Appendix A Character sets.

ASCII hex.	ASCII graph	RA68 graph	Revised Report symbol	CDC graph	CDC octal
00-1F			typographical display feature		
20	space	space	space symbol	space	55
21	!		style tally monad symbol	=	64
22	"	"	quote symbol	=	64
23		#	style two comment symbol	=	64
24	$	$	formatter symbol	$	53
25	%	%	percent symbol		00
26	&	&	ampersand symbol	^	67
27	'	'	apostrophe	'	70
28	(	(	open, brief begin symbol	(	51
29	)	)	close, brief end symbol	)	52
2A	*	*	asterisk symbol	*	47
2B	+	+	plus symbol	+	45
2C	,	,	comma symbol	,	56
2D	-	-	minus symbol	-	46
2E	.	.	point symbol	.	57
2F	/	/	divided by symbol	/	50
30-39	0-9	0-9	digit zero - nine symbol	0-9	33-44
3A	:	:	colon-, up to-, label-, routine-symbol	:	63
3B	;	;	go on symbol	;	77
3C	<	<	less than symbol	<	72
3D	=	=	equals-, is defined as-symbol	=	54
3E	>	>	greater than symbol	>	73
3F	?		style tally monad symbol		71
40	@	@	at symbol	@	74
41-5A	A-Z	A-Z	letter a - letter z symbol	A-Z	01-32
5B	[	[	brief sub symbol	[	61
5C			times ten to the power symbol	'	70
5D	]	]	brief bus symbol	]	62
5E			not symbol	¬	76
5F	_	_	underscore	_	65
60-7A	a-z	a-z	style one letter a-z symbol		
7B			style tally monad symbol		
7C	|		brief in-,out-,else-, then-symbol	|	66
7D			style tally monad symbol		
7E			style tally monad symbol		
7F			letter aleph		
80-FF			style tally monad symbol		
100-FFF			other string item		

Appendix B Compilation control statement parameters.

The compiler may be invoked by a statement of one of the forms:

A68(P_1,P_2,...,P_k) or A68(0,P_1,P_2,...,P_k).

The latter form allows the use of letter aleph in indicators. This form is intended for the compilation of the standard circumludes only.

Each parameter defines one or more entities to be used during compilation; these entities are of four kinds: boolean, file, library/file combination or enumerated.

Each parameter consists of a one or two letter keyword optionally followed by an equals sign and a value of the appropriate kind:

for boolean parameters the only value allowed is "0" meaning false.

for file parameters the value takes the form of an identifier

for library/file parameters, a single identifier or two identifiers separated by a "/" or"-" , possibly followed by a "-" and an integer value are possible.

for enumerated the value is a mnemonic.

For each parameter keyword two defaults exist; the first applies when the parameter is omitted, the second when only the keyword is given.

The available parameters are listed below in the form:

```
"keyword: : description of entity defined
            ("type":"first default"/second default")

A  : subscript checking  (boolean : on/off)
B  : binary output file (file : LGO/LGO)
C  : source line width (integer :72/80)
D  : symbolic dump information (boolean :off/on)
F  : stropping (enumerated(STROP, FLAG, POINT, UPPER, RES):
                           FLAG or POINT/STROP)
FA : FORTRAN style allocation (boolean : FA(as circumlude/on)
G  : circumlude (file-integer : P-parameter/*) or
                 (library/file-level)
I  : source file (file : INPUT/COMPILE)
J  : input and listing in ASCII 12-bit code (boolean : off/on)
K  : source listing in ASCII 12-bit code (boolean : off/on)
L  : source listing file (file: OUTPUT/OUTPUT)
M  : error listing file (file: L-parameter/*)
N  : circumlude output overlay file
             (file-integer: none/B-parameter)
O  : object code listing file (file: none/L-parameter)
P  : circumlude (enumerated(0, LONG, TORBAS)
                                 : ( standard/standard)
               or (library-file) or (file)  :
S  : append status info to source line (boolean: off/on)
T  : generate code with fatal errors (boolean: off/on)
Z  : instruction scheduling (boolean: off/on)
```

Appendix C. Grammar.

C.1 Terminal symbols.

Terminal symbols needed for ALGOL 68 are:

```
OPEN   , CLOSE , COMMA , COLON , GOON   , EQUALS, AT     , IS    ,
SUB    , BUS   , STICK , SKIP  , OF     , NIL   , FORMAL, AGAIN,
TAG    , DO    , OD    , QUAL  , STRUCT, UNION , MODE   , OP    ,
PRIO   , PROC  , FLEX  , REF   , VOID   , BEGIN , WHILE , IF    ,
CASE   , OUSE  , ELIF  , ESAC  , FI     , END   , THEN  , ELSE  ,
IN     , OUT   , PAR   , GOTO  , FOR    , FROM  , BY    , TO    ,
PRIO-CYPHER    , OP-INDICANT    , MODE-INDICANT , COUNT-FRAME  ,
LITERAL-FRAME  , GENERAL-FRAME  , OPEN-SLASH    , SLASH-CLOSE  ,
DENOTATION     , BECOMES        , COMPLETER     , DYADIC       ,
MONADIC        , DOLLAR         .
```

Additionally the following termainal symbols are required to support the language extensions:

INLINE, XREF, XDEF, FEDX, PROG, FORTRAN, DESCRIPTOR, END-DESCR.

C.2 GRAMMAR for Cyber ALGOL 68.

```
ALGOL-68:
#1# TAG(NAME1) ?64, COLON ?65, ENCLOSED-CLAUSE /$PRELUDE(NAME1);
#2# PROG ?65, PARTICULAR-PROGRAM(N1) ?99, GORP /$PARTP(N1).
ENCLOSED-CLAUSE:
#1# BEGIN ?3, SERIAL-CLAUSE(LABELS1) ?7, END /$RANGE;
#2# OPEN ?4, SERIAL-CLAUSE(LABELS1) ?27, CLOSE /$RANGE;
#3# BEGIN ?3, UNITS-LIST(N1) ?8, END /$COLL(N1);
#4# OPEN ?4, UNITS-LIST(N1) ?8, CLOSE /$COLL(N1);
#5# PAR, $OUTPNTR($PARNAMX) ?28, BEGIN ?60, PAR-UNITS-LIST(N1)
    ?91, END /$PARCL(N1);
#6# PAR, $OUTPNTR($PARNAMX) ?28, OPEN ?60, PAR-UNITS-LIST(N1)
    ?93, CLOSE /$PARCL(N1);
#7# IF ?5, SERIAL-CLAUSE(LABELS1) ?29, THEN ?61,
    SERIAL-CLAUSE(LABELS2), $RANGE ?116,CONDITION-ELSE-CLAUSE
    ?133, FI /$IFCL(LABELS1);
#8# OPEN ?4, SERIAL-CLAUSE(LABELS1) ?27, STICK ?59,
     SERIAL-CLAUSE(LABELS2), $RANGE ?114, CONDITION-STICK-CLAUSE
     ?131, CLOSE /$IFCL(LABELS1);
#9# CASE ?6, SERIAL-CLAUSE(LABELS1) ?31, IN ?62, UNITS-LIST(N1)
     94, CASE-OUT-CLAUSE ?117, ESAC /$CASECL(LABELS1,N1);
#10# OPEN ?4, SERIAL-CLAUSE(LABELS1) ?27, STICK ?59,
     UNITS-LIST(N1) ?89, $CORRAL ?114, CASE-STICK-CLAUSE ?131,
     CLOSE /$CASECL(LABELS1,N1);
#11# CASE ?6, SERIAL-CLAUSE(LABELS1) ?31, IN ?62,
     SPECIFIED-UNIT-LIST(N1) ?95, CONFORMITY-OUT-CLAUSE ?117,
     ESAC /$COCACL(LABELS1,N1);
#12# OPEN ?4, SERIAL-CLAUSE(LABELS1) ?27, STICK ?59,
     SPECIFIED-UNIT-LIST(N1), $CORRAL ?114,
     CONFORMITY-STICK-CLAUSE ?131, CLOSE /$COCACL(LABELS1,N1);
```

```
#13# OPEN ?4, CLOSE /$VACUUM;
#14# BEGIN ?3, END /$VACUUM;
#15# FOR-PART(NAME1) ?23, FROM-PART(N1) ?51, BY-PART(N2) ?81,
     TO-PART(N3),   $FBTS(NAME1,N1,N2,N3)   ?127,   WHILE-PART(N4)
     ?142, DO  ?153, SERIAL-CLAUSE(LABELS1) ?156, OD
     /$DOLOOP(NAME1,N4).
SERIAL-CLAUSE(LABELS):
#1# UNIT(NONE) ?30 /*SET0(LABELS);
#2# SERIAL(EXITS1,GOONS1,LABELS1) ?10, UNIT(NONE) ?30,
    /$SERCL2(LABELS,EXITS1,GOONS1,LABELS1).
SERIAL(EXITS,GOONS,LABELS):
#1# UNIT(NONE) ?9, GOON /$SERUN2(EXITS,GOONS,LABELS,0,0,0);
#2# TAG(NAME1) ?11, COLON /$SERLA2(EXITS,GOONS,LABELS,0,0,NAME1);
#3# DECLARATIONS-LIST(N1) ?12, GOON
    /$SERDA2(EXITS,GOONS,LABELS,0,0,0,N1);
#4# UNIT(NONE) ?9, COMPLETER ?33, TAG(NAME1) ?64, COLON
    /$SERCO2(EXITS,GOONS,LABELS,0,0,NAME1);
#5# SERIAL(EXITS1,GOONS1,LABELS1) ?10, UNIT(NONE) ?30, GOON
    /$SERUN2(EXITS,GOONS,LABELS,EXITS1,GOONS1,LABELS1);
#6# SERIAL(EXITS1,GOONS1,LABELS1) ?10, TAG(NAME1) ?11, COLON
    /$SERLA2(EXITS,GOONS,LABELS,EXITS1,GOONS1,NAME1);
#7# SERIAL(EXITS1,GOONS1,LABELS1) ?10, DECLARATIONS-LIST(N1) ?12,
    GOON /$SERDA2(EXITS,GOONS,LABELS,EXITS1,GOONS1,LABELS1,N1);
#8# SERIAL(EXITS1,GOONS1,LABELS1) ?10, UNIT(NONE) ?30,
    COMPLETER ?33, TAG(NAME1) ?64, COLON
    /$SERCO2(EXITS,GOONS,LABELS,EXITS1,GOONS1,NAME1);
#9# SERIAL(EXITS1,GOONS1,LABELS1) ?10, PROG
    /$SERPA1(EXITS,GOONS,LABELS,EXITS1,GOONS1,LABELS1).
PARTICULAR-PROGRAM(N):
#1# ENCLOSED-CLAUSE /*SET0(N);
#2# TAG(NAME1)  ?64, COLON, $PPLAB(NAME1) ?65,
    PARTICULAR-PROGRAM(N1) /*COUNT1(N,N1).
CONDITION-ELSE-CLAUSE:
#1# /$SKIP;
#2# ELSE ?134, SERIAL-CLAUSE(LABELS1) /$RANGE;
#3# ELIF ?135, SERIAL-CLAUSE(LABELS1) ?148, THEN ?61,
    SERIAL-CLAUSE(LABELS2), $RANGE ?116, CONDITION-ELSE-CLAUSE
    /$IFCL(LABELS1).
CONDITION-STICK-CLAUSE:
#1# /$SKIP;
#2# STICK ?106, SERIAL-CLAUSE(LABELS1) /$RANGE;
#3# AGAIN ?132, SERIAL-CLAUSE(LABELS1) ?145, STICK ?106,
    SERIAL-CLAUSE(LABELS2), $RANGE ?114, CONDITION-STICK-CLAUSE
    /$IFCL(LABELS1).
CASE-OUT-CLAUSE:
#1# /$SKIP;
#2# OUT ?118, SERIAL-CLAUSE(LABELS1) /$RANGE;
#3# OUSE ?119, SERIAL-CLAUSE(LABELS1) ?136, IN ?149,
    UNITS-LIST(N1) ?94, CASE-OUT-CLAUSE /$CASECL(LABELS1,N1).
CASE-STICK-CLAUSE:
#1# /$SKIP;
#2# STICK ?106, SERIAL-CLAUSE(LABELS1) /$RANGE;
#3# AGAIN ?132, SERIAL-CLAUSE(LABELS1) ?145, STICK ?110,
    UNITS-LIST(N1) ?89, $CORRAL ?114, CASE-STICK-CLAUSE
    /$CASECL(LABELS1,N1).
```

```
CONFORMITY-OUT-CLAUSE:
#1#  /$SKIP;
#2# OUT ?118, SERIAL-CLAUSE(LABELS1) /$RANGE;
#3# OUSE ?119, SERIAL-CLAUSE(LABELS1) ?136, IN ?150,
    SPECIFIED-UNIT-LIST(N1) ?95, CONFORMITY-OUT-CLAUSE
    /$COCACL(LABELS1,N1).
CONFORMITY-STICK-CLAUSE:
#1# /$SKIP;
#2# STICK ?106, SERIAL-CLAUSE(LABELS1) /$RANGE;
#3# AGAIN ?132, SERIAL-CLAUSE(LABELS1) ?145, STICK ?150,
    SPECIFIED-UNIT-LIST(N1) ?90, $CORRAL ?114,
    CONFORMITY-STICK-CLAUSE /$COCACL(LABELS1,N1).
DECLARATIONS-LIST(N):
#1# MODE-DEFINITIONS(N1) ?14 /*COPY(N,N1);
#2# PRIO-DEFINITIONS ?13 /*SET0(N);
#3# CONSTANT-DEFINITIONS(N1,MODE1) ?15 /*COPY(N,N1);
#4# PROCEDURE-DEFINITIONS(N1) ?16 /*COPY(N,N1);
#5# OP-MODE-DEFINITIONS(N1,PROCM1) ?17 /*COPY(N,N1);
#6# OP-ROUT-DEFINITIONS(N1) ?17 /*COPY(N,N1);
#7# VARIABLE-DECLARATIONS(N1,MODE1) ?18 /$VDECS1(N,N1);
#8# PROCEDURE-DECLARATIONS(N1,QUAL1) ?16 /*COPY(N,N1);
#9# DECLARATIONS-LIST(N1) ?12, COMMA ?34,
    MODE-DEFINITIONS(N2) ?14  /*ADD(N,N1,N2);
#10# DECLARATIONS-LIST(N1) ?12, COMMA ?34,
     PRIO-DEFINITIONS ?13  /*COPY(N,N1);
#11# DECLARATIONS-LIST(N1) ?12, COMMA ?34,
     CONSTANT-DEFINITIONS(N2,MODE1) ?15 /*ADD(N,N1,N2);
#12# DECLARATIONS-LIST(N1) ?12, COMMA ?34,
     PROCEDURE-DEFINITIONS(N2) ?16 /*ADD(N,N1,N2);
#13# DECLARATIONS-LIST(N1) ?12, COMMA ?34,
     OP-MODE-DEFINITIONS(N2,PROCM2) ?17 /*ADD(N,N1,N2);
#14# DECLARATIONS-LIST(N1) ?12, COMMA ?34,
     OP-ROUT-DEFINITIONS(N2) ?17 /*ADD(N,N1,N2);
#15# DECLARATIONS-LIST(N1) ?12, COMMA ?34,
     VARIABLE-DECLARATIONS(N2,MODE1) ?18 /$VDECS2(N,N1,N2);
#16# DECLARATIONS-LIST(N1) ?12, COMMA ?34,
     PROCEDURE-DECLARATIONS(N2,QUAL1) ?16 /*ADD(N,N1,N2).
UNITS-LIST(N):
#1# UNIT(NONE1) ?96, COMMA ?32, UNIT(NONE2) /*SET2(N);
#2# UNITS-LIST(N1) ?8, COMMA ?32, UNIT(NONE) /*COUNT1(N,N1).
PAR-UNITS-LIST(N):
#1# UNIT(NONE) /$PROCV2(N);
#2# PAR-UNITS-LIST(N1) ?91, COMMA ?92, UNIT(NONE) /$PROCV3(N,N1).
SPECIFIED-UNIT-LIST(N):
#1#  OPEN ?4, $NEWDCL(F1,STAT1) ?147, MOID(MODE1) ?155, CLOSE
     ?146, COLON, $OLDFIL(F1,STAT1) ?157, UNIT(NONE)
    /$SPECV2(N,0,MODE1);
#2# OPEN ?4, $NEWDCL(F1,STAT1) ?147, DECLARER(MODE1) ?155,
    TAG(NAME1) ?155, CLOSE ?146, COLON,
    $SPEC1(F1,STAT1,MODE1,NAME1) ?157, UNIT(NONE)
    /$CONF2(N,0,MODE1);
#3# SPECIFIED-UNIT-LIST(N1) ?90, COMMA ?115, OPEN, $NEWFOR(STAT1)
    ?147, MOID(MODE1) ?155, CLOSE ?146, COLON, $OLDDCL(STAT1)
    ?157, UNIT(NONE) /$SPECV2(N,N1,MODE1);
#4# SPECIFIED-UNIT-LIST(N1) ?90, COMMA ?115, OPEN, $NEWFOR(STAT1)
```

```
    ?147, DECLARER(MODE1) ?155, TAG(NAME1) ?155, CLOSE ?146,
    COLON, $SPEC2(STAT1,MODE1,NAME1) ?157, UNIT(NONE)
    /$CONF2(N,N1,MODE1).
MODE-DEFINITIONS(N):
#1# MODE ?19, INDICANT(NAME1), $MODEF1(NAME1,STAT1) ?69,
    EQUALS ?122, VODE(MODE1) /$MODEF3(N,0,NAME1,STAT1,MODE1);
#2# MODE-DEFINITIONS(N1) ?14, COMMA ?34, INDICANT(NAME1),
    $MODEF1(NAME1,STAT1) ?69, EQUALS ?122, VODE(MODE1)
    /$MODEF3(N,N1,NAME1,STAT1,MODE1).
PRIO-DEFINITIONS:
#1# PRIO ?20, PRIO-OP(NAME1) ? 43, EQUALS ?104, PRIO-CYPHER
    /$PRIODEF(NAME1);
#2# PRIO-DEFINITIONS ?13, COMMA ?34, PRIO-OP(NAME1) ?43,
    EQUALS ?104, PRIO-CYPHER /$PRIODEF(NAME1).
CONSTANT-DEFINITIONS(N,MODE):
#1# $NEWDCL(F1,STAT1) ?49, DECLARER(MODE1) ?100, TAG(NAME1) ?70,
    EQUALS, $CONSD1(F1,STAT1,MODE1,NAME1) ?138,
    EXT-UNIT-3(NAME2)
/$DEF2(NAME1,N,MODE,0,MODE1,NAME2,QIL2OP"IDCL");
#2# CONSTANT-DEFINITIONS(N1,MODE1) ?15, COMMA ?34, TAG(NAME1)
    ?67, EQUALS,$IDENT(MODE1,NAME1) ?138,EXT-UNIT-3(NAME2)
    /$DEF2(NAME1,N,MODE,N1,MODE1,NAME2,QIL2OP"IDCL").
PROCEDURE-DEFINITIONS(N):
#1# PROC ?50, PROC-TAG(NAME1) ?45, EQUALS ?77,
    FORMAL-PLAN(PROC1),
    $PROCD1(PROC1,NAME1) ?78, EXT-UNIT-2(NAME2,N1)
    /$DEF1(NAME1,N,0,PROC1,NAME2,QIL2OP"IDCL",N1);
#2# PROCEDURE-DEFINITIONS(N1) ?16, COMMA ?34, PROC-TAG(NAME1)
    ?45, EQUALS ?77, FORMAL-PLAN(PROC1), $PROCD1(PROC1,NAME1)
    ?78, EXT-UNIT-2(NAME2,N2)
    /$DEF1(NAME1,N,N1,PROC1,NAME2,QIL2OP"IDCL",N2).
OP-MODE-DEFINITIONS(N,PROCM):
#1# OP ?21, VIR-PAR-PLAN(PROC1) ?46, OP-OP(TYPE1,NAME1) ?47,
    EQUALS, $OPDEF1(PROC1,TYPE1,NAME1) ?139, EXT-UNIT(NAME2)
    /$DEF2(NAME1,N,PROCM,0,PROC1,NAME2,QIL2OP"OPDCL");
#2# OP ?21, VIR-PAR-PLAN(PROC1) ?46, OP-OP(TYPE1,NAME1) ?47,
    EQUALS, $OPDEF1(PROC1,TYPE1,NAME1) ?139, INLINE(NAME2) ?152,
    SKIP /$OPDEF5(PROCM,N,PROC1,0,NAME1,NAME2);
#3# OP-MODE-DEFINITIONS(N1,PROCM1) ?17, COMMA ?34,
    OP-OP(TYPE1,NAME1) ?47, EQUALS,
    $OPDEF1(PROCM1,TYPE1,NAME1) ?139, EXT-UNIT(NAME2)
    /$DEF2(NAME1,N,PROCM,N1,PROCM1,NAME2,QIL2OP"OPDCL");
#4# OP-MODE-DEFINITIONS(N1,PROCM1) ?17, COMMA ?34,
    OP-OP(TYPE1,NAME1) ?47, EQUALS,
    $OPDEF1(PROCM1,TYPE1,NAME1) ?139, INLINE(NAME2) ?152, SKIP
    /$OPDEF5(PROCM,N,PROCM1,N1,NAME1,NAME2).
OP-ROUT-DEFINITIONS(N):
#1# OP ?21, OP-OP(TYPE1,NAME1) ?47, EQUALS ?107,
    FORMAL-PLAN(PROC1),
    $OPDEF1(PROC1,TYPE1,NAME1) ?139, EXT-UNIT-3(NAME2)
    /$DEF1(NAME1,N,0,PROC1,NAME2,QIL2OP"OPDCL",0);
#2# OP-ROUT-DEFINITIONS(N1) ?17, COMMA ?34, OP-OP(TYPE1,NAME1)
    ?47, EQUALS ?107, FORMAL-PLAN(PROC1),
    $OPDEF1(PROC1,TYPE1,NAME1) ?139, EXT-UNIT-3(NAME2)
    /$DEF1(NAME1,N,N1,PROC1,NAME2,QIL2OP"OPDCL",0).
```

```
VARIABLE-DECLARATIONS(N,MODE):
#1# $NEWDCL(F1,STAT1) ?49, DECLARER(MODE1) ?100, TAG(NAME1) ?70
    /$VARD4(N,MODE,QIL2OP"LOC",STAT1,MODE1,NAME1);
#2# $NEWDCL(F1,STAT1) ?49, DECLARER(MODE1) ?100, TAG(NAME1) ?70,
    BECOMES, $VARD5(QIL2OP"LOC",STAT1,MODE1,NAME1) ?121,
    UNIT(NONE) /$VARD7(N,MODE,0,MODE1);
#3# QUAL(QUAL1), $NEWACT(STAT1) ?22, DECLARER(MODE1) ?35,
    TAG(NAME1) ?68 /$VARD4(N,MODE,QUAL1,STAT1,MODE1,NAME1);
#4# QUAL(QUAL1), $NEWACT(STAT1) ?22, DECLARER(MODE1) ?35,
    TAG(NAME1) ?68, BECOMES, $VARD5(QUAL1,STAT1,MODE1,NAME1)
    ?121, UNIT(NONE) /$VARD7(N,MODE,0,MODE1);
#5# VARIABLE-DECLARATIONS(N1,MODE1) ?18, COMMA ?34, TAG(NAME1)
    ?70 /$VARD6(N,MODE,N1,MODE1,NAME1);
#6# VARIABLE-DECLARATIONS(N1,MODE1) ?18, COMMA ?34, TAG(NAME1)
    ?70, BECOMES, $VARD1(MODE1,NAME1) ?121, UNIT(NONE)
    /$VARD7(N,MODE,N1,MODE1).
PROCEDURE-DECLARATIONS(N,QUAL):
#1# PROC ?50, PROC-TAG(NAME1) ?45, BECOMES ?77,
    FORMAL-PLAN(PROC1),
    $PROCD7(QIL2OP"LOC",NAME1,PROC1) ?78, DESCRIPTION(N1) ?159,
    UNIT(NONE) /$PROCD8(N,QUAL,0,QIL2OP"LOC",PROC1,N1);
#2# QUAL(QUAL1), $NEWACT(STAT1) ?22, PROC ?50, PROC-TAG(NAME1)
    ?68, BECOMES, $OLDDCL(STAT1) ?77, FORMAL-PLAN(PROC1),
    $PROCD7(QUAL1,NAME1,PROC1) ?78, DESCRIPTION(N1) ?159,
    UNIT(NONE) /$PROCD8(N,QUAL,0,QUAL1,PROC1,N1);
#3# PROCEDURE-DECLARATIONS(N1,QUAL1) ?16, COMMA ?34,
    PROC-TAG(NAME1) ?68, BECOMES ?77, FORMAL-PLAN(PROC1) ?78,
    $PROCD7(QUAL1,NAME1,PROC1) ?78, DESCRIPTION(N2) ?159,
    UNIT(NONE) /$PROCD8(N,QUAL,N1,QUAL1,PROC1,N2).
VODE(MODE):
#1# VOID /$VODERR(MODE);
#2# DECLARER(MODE1) /*COPY(MODE,MODE1).
MODES-LIST(LIST,N):
#1# VODE(MODE1) /$MODSL2(LIST,N,0,0,MODE1);
#2# MODES-LIST(LIST1,N1) ?125, COMMA ?102, VODE(MODE1)
 /$MODSL2(LIST,N,LIST1,N1,MODE1).
MOID(MODE):
#1# VOID /$MOID1(MODE);
#2# DECLARER(MODE1) ?35 /*COPY(MODE,MODE1).
MOVO-LIST(LIST,N):
#1# MOVO(MODE1) /$MODSL2(LIST,N,0,0,MODE1);
#2# MOVO-LIST(LIST1,N1) ?125, COMMA ?49, MOVO(MODE1)
    /$MODSL2(LIST,N,LIST1,N1,MODE1).
MOVO(MODE):
#1# VOID /$MOID2(MODE);
#2# DECLARER(MODE1) /*COPY(MODE,MODE1).
DECLARER(MODE):
#1# MODE-INDICANT(NAME1) /$DECL1(MODE,NAME1,0);
#2# FLEX ?38, MODE-INDICANT(NAME1) /$DECL1(MODE,NAME1,1);
#3# REF, $INFER(STAT1) ?72, DECLARER(MODE1)
    /$DECL2(MODE,STAT1,MODE1);
#4# ROWER(N1), $BESTOW(STAT1) ?73, DECLARER(MODE1)
    /$DECL3(MODE,N1,STAT1,MODE1);
#5# FLEX ?38, ROWER(N1), $BESTOW(STAT1) ?73, DECLARER(MODE1)
    /$DECL4(MODE,N1,STAT1,MODE1);
```

```
#6# STRUCT ?39, OPEN, $BESTOW(STAT1) ?101,
    PORTRAYERS-LIST(LIST1,MODE1,FIELDS1,SELECTS1,PORTRAYS1) ?124,
    CLOSE /$DECL5(MODE,STAT1,LIST1,FIELDS1,SELECTS1,PORTRAYS1);
#7# UNION ?40, OPEN, $NEWVIR(STAT1) ?102, MOVO-LIST(LIST1,N1)
    ?125, CLOSE /$DECL6(MODE,STAT1,LIST1,N1);
#8# PROC ?36, VIRTUAL-PLAN(PROC1) /$DECL7(MODE,PROC1).
ROWER(N):
#1# SUB ?41, BOUNDS-LIST(N1) ?74, BUS /$ROW(N,N1);
#2# OPEN-SLASH ?42, BOUNDS-LIST(N1) ?74, SLASH-CLOSE /$ROW(N,N1).
BOUNDS-LIST(N):
#1# BOUNDS(BNDS1) /$BOUNDS2(N,0,BNDS1);
#2# BOUNDS-LIST(N1) ?74, COMMA ?103, BOUNDS(BNDS1)
    /$BOUNDS2(N,N1,BNDS1).
BOUNDS(BNDS):
#1# BOUND(BND1) ?75 /$BOUNDS3(BNDS,BND1);
#2# BOUND(BND1) ?75, COLON ?103, BOUND(BND2)
    /$BOUNDS4(BNDS,BND1,BND2).
BOUND(BND):
#1#  /$BOUND1(BND);
#2# UNIT(NONE) /$BOUND2(BND).
PORTRAYERS-LIST(LIST,MODE,FIELDS,SELECTS,PORTRAYS):
#1# DECLARER(MODE1) ?123, TAG(NAME1)
 /$PORT1(LIST,MODE,FIELDS,SELECTS,PORTRAYS,MODE1,NAME1);
#2# PORTRAYERS-LIST(LIST1,MODE1,FIELDS1,SELECTS1,PORTRAYS1) ?124,
    COMMA ?140, TAG(NAME1)
    /$PORT2(LIST,MODE,FIELDS,SELECTS,PORTRAYS,LIST1,MODE1,FIELDS1,
    SELECTS1,PORTRAYS1,NAME1);
#3# PORTRAYERS-LIST(LIST1,MODE1,FIELDS1,SELECTS1,PORTRAYS1) ?124,
    COMMA ?140, $PORT3(SELECTS1,PORTRAYS1) ?101,
    DECLARER(MODE2) ?123, TAG(NAME1)
    /$PORT2(LIST,MODE,FIELDS,SELECTS,PORTRAYS,LIST1,MODE2,
    FIELDS1,SELECTS1,PORTRAYS1,NAME1).
FORMAL-PLAN(PROC):
#1# $NEWDCL(F1,STAT1) ?137, MOID(MODE1) ?126, COLON
    /$FORPL1(PROC,F1,STAT1,MODE1);
#2# OPEN ?105, FORMAL, $NEWFOR(STAT1) ?63,
    PARAMETERS-LIST(MODE1,PARMS1) ?98, CLOSE, $FORVIR ?137,
    MOID(MODE2) ?126, COLON /$FORPL2(PROC,STAT1,PARMS1,MODE2).
PARAMETERS-LIST(MODE,PARMS):
#1# DECLARER(MODE1) ?97, TAG(NAME1)
    /$PARML2(MODE,PARMS,MODE1,0,NAME1);
#2# PARAMETERS-LIST(MODE1,PARMS1) ?98, COMMA ?120, TAG(NAME1)
    /$PARML2(MODE,PARMS,MODE1,PARMS1,NAME1);
#3# PARAMETERS-LIST(MODE1,PARMS1) ?98, COMMA ?120,DECLARER(MODE2)
    ?97, TAG(NAME1) /$PARML2(MODE,PARMS,MODE2,PARMS1,NAME1).
VIRTUAL-PLAN(PROCM):
#1# $NEWVIR(STAT1) ?141, MOID(MODE1) /$VIRPL1(PROCM,STAT1,MODE1);
#2# VIR-PAR-PLAN(PROC1) /*COPY(PROCM,PROC1).
VIR-PAR-PLAN(PROCM):
#1# OPEN, $NEWVIR(STAT1) ?102, MODES-LIST(LIST1,N1) ?125, CLOSE,
    $CORRAL ?141, MOID(MOID1)
    /$VIRPL2(PROCM,STAT1,LIST1,N1,MOID1).
FOR-PART(NAME):
#1#  /*SET0(NAME);
#2# FOR ?24, TAG(NAME1) /*COPY(NAME,NAME1).
```

```
FROM-PART(N):
#1#  /*SET0(N);
#2# FROM ?52, UNIT(NONE) /$FBT1(N,QIL2OP"FROM").
BY-PART(N):
#1#  /*SET0(N);
#2# BY ?82, UNIT(NONE) /$FBT1(N,QIL2OP"BY").
TO-PART(N):
#1#  /*SET0(N);
#2# TO ?109, UNIT(NONE) /$FBT1(N,QIL2OP"TO").
WHILE-PART(N):
#1#  /*SET0(N);
#2# WHILE ?143, SERIAL-CLAUSE(LABELS1) /$WHILEP(N,LABELS1).
EXT-UNIT(NAME):
#1# NON-ROUT /*SET0(NAME);
#2# FORMAL-PLAN(PROC1) ?48, UNIT(NONE) /$ROUT2(NAME,PROC1,0);
#3# XREF(NAME1) ?80, SKIP /$XREF1(NAME,NAME1,0);
#4# XDEF(NAME1) ?79, UNIT(NONE) ?108, FEDX /*COPY(NAME,NAME1).
EXT-UNIT-2(NAME,N):
#1# DESCRIPTION(N1) ?159, UNIT(PROC1) /*COPY2(NAME,PROC1,N,N1);
#2# FORTRAN(NAME1) ?80, SKIP /$FTN(NAME,N,NAME1);
#3# XREF(NAME1) ?80, SKIP /$XREF1(NAME,NAME1,N);
#4# DESCRIPTION(N1) ?159, XDEF(NAME1) ?79, UNIT(NONE) ?108, FEDX
    /*COPY2(NAME,NAME1,N,N1).
EXT-UNIT-3(NAME):
#1# UNIT(PROC1) /*COPY(NAME,PROC1);
#2# XREF(NAME1) ?80, SKIP /$XREF1(NAME,NAME1,0);
#3# XDEF(NAME1) ?79, UNIT(NONE) ?108, FEDX /*COPY(NAME,NAME1).
UNIT(NAME):
#1# NON-ROUT /*SET0(NAME);
#2# FORMAL-PLAN(PROC1) ?48, DESCRIPTION(N1) ?48, UNIT(NAME1)
 /$ROUT2(NAME,PROC1,N1).
NON-ROUT:
#1# TERTIARY ?25;
#2# TERTIARY ?25, IS(IS1) ?54, TERTIARY /$OUTOPER(IS1,2);
#3# TERTIARY ?25, BECOMES ?53, UNIT(NONE)
    /$OUTOPER(QIL2OP"ASSIGN",2).
TERTIARY:
#1# FORMULA(N1) ?25 /$DYADIC(N1).
FORMULA(N):
#1# FORMULA(N1) ?25, APL-OP(TYPE1,NAME1), $DYAD(NAME1) ?83,
    OPERAND /*COUNT1(N,N1);
#2# OPERAND /*SET0(N).
OPERAND:
#1# APL-OP(TYPE1,NAME1) ?44, OPERAND /$MONADIC(TYPE1,NAME1);
#2# SECONDARY.
SECONDARY:
#1# PRIMARY ?158;
#2# QUAL(QUAL1), $NEWACT(STAT1) ?22, DECLARER(MODE1) ?35
    /$GENTOR(QUAL1,STAT1,MODE1);
#3# TAG(NAME1) ?11, OF, $OUTPNTR(NAME1) ?66, SECONDARY
    /$OUTOPER(QIL2OP"OF",2).
PRIMARY:
#1# DENOTATION(CONS1) /$OUTPNTR(CONS1);
#2# NIL /$OUTOPER(QIL2OP"NIL",0);
#3# SKIP /$SKIP;
```

```
#4# GOTO ?26, TAG(NAME1) /$JUMP(NAME1);
#5# TAG(NAME1) /$OUTPNTR(NAME1);
#6# ENCLOSED-CLAUSE;
#7# $NEWDCL(F1,STAT1) ?49, MOID(MODE1) ?37,
    $CAST1(F1,STAT1,MODE1)
    ?71, ENCLOSED-CLAUSE /$OUTOPER(QIL2OP"CAST",2);
#8# PRIMARY ?158, SUB ?57, INDEXER-LIST(SUBS1,SLICS1) ?85, BUS
    /$SLICE(SUBS1,SLICS1);
#9# PRIMARY ?158, OPEN-SLASH ?57, INDEXER-LIST(SUBS1,SLICS1) ?88,
    SLASH-CLOSE /$SLICE(SUBS1,SLICS1);
#10# PRIMARY ?158, OPEN ?56, UNIT-LIST(N1) ?84, CLOSE /$CALL(N1);
#11# DOLLAR, $OUTPNTR($FORMATX), DYNAMIC-SEQUENCE(N1) ?58, COMMA
     ?58,DENOTATION(CONS1) ?129, DOLLAR /$FORMAT2(N1,CONS1).
UNIT-LIST(N):
#1# UNIT(NONE) /*SET1(N);
#2# UNIT-LIST(N1) ?84, COMMA ?56, UNIT(NONE) /*COUNT1(N,N1).
INDEXER-LIST(SUBS,SLICS):
#1# INDEXER(SLICS1) /$INDEX2(SUBS,SLICS,0,SLICS1,0);
#2# INDEXER-LIST(SUBS1,SLICS1), COMMA ?57, INDEXER(SLICS2)
    /$INDEX2(SUBS,SLICS,SUBS1,SLICS1,SLICS2).
INDEXER(SLICS):
#1# UNIT(NONE) ?85 /*SET0(SLICS);
#2# NEW-BOUND(BND1) /$INDEX3(SLICS,BND1);
#3# OLD-BOUND(BND1) ?86, COLON, $LWB(BND1) ?128, OLD-BOUND(BND2),
    $UPB(BND2) ?154, NEW-BOUND(BND3)
    /$INDEX4(SLICS,BND1,BND2,BND3).
NEW-BOUND(BND):
#1#  /*SET0(BND);
#2# AT ?87, UNIT(NONE) /$AT(BND).
OLD-BOUND(BND):
#1#  /*SET0(BND);
#2# UNIT(NONE) ?85 /*SET1(BND).
PROC-TAG(NAME):
#1# TAG(NAME1) /$PROCTAG(NAME,NAME1).
APL-OP(TYPE,NAME):
#1# EQUALS /$APLEQU(TYPE,NAME);
#2# DYADIC(NAME1) /$APLDYA(TYPE,NAME,NAME1);
#3# MONADIC(NAME1) /$APLMON(TYPE,NAME,NAME1);
#4# OP-INDICANT(NAME1) /$APLMON(TYPE,NAME,NAME1).
PRIO-OP(NAME):
#1# EQUALS /$PRIOEQU(NAME);
#2# DYADIC(NAME1) /$PRIONAM(NAME,NAME1);
#3# MONADIC(NAME1) /$PRIONAM(NAME,NAME1);
#4# OP-INDICANT(NAME1) /$PRIONAM(NAME,NAME1);
#5# MODE-INDICANT(NAME1) /$PRIONAM(NAME,NAME1).
OP-OP(TYPE,NAME):
#1# EQUALS /$OPEQU(TYPE,NAME);
#2# DYADIC(NAME1) /$OPDYA(TYPE,NAME,NAME1);
#3# MONADIC(NAME1) /$OPMON(TYPE,NAME,NAME1);
#4# OP-INDICANT(NAME1) /$OPMON(TYPE,NAME,NAME1);
#5# MODE-INDICANT(NAME1) /$OPMON(TYPE,NAME,NAME1).
INDICANT(NAME):
#1# OP-INDICANT(NAME1) /*COPY(NAME,NAME1);
#2# MODE-INDICANT(NAME1) /*COPY(NAME,NAME1).
DYNAMIC-SEQUENCE(N):
```

```
#1#  /*SET0(N);
#2# DYNAMIC-SEQUENCE(N1) ?58, DYNAMIC /*COUNT1(N,N1).
DYNAMIC:
#1# COUNT-FRAME ?111, DENOTATION(CONS1) /$DYNAM1(CONS1);
#2# COUNT-FRAME ?111, ENCLOSED-CLAUSE
    /$OUTOPER(QIL2OP"GENERAL",1);
#3# LITERAL-FRAME ?112, DENOTATION(CONS1) /$DYNAM3(CONS1);
#4# LITERAL-FRAME ?112, ENCLOSED-CLAUSE
    /$OUTOPER(QIL2OP"LITERAL",1);
#5# GENERAL-FRAME ?113, OPEN ?130, UNIT(NONE) ?144,
    CLOSE /$DYNAM5(1);
#6# GENERAL-FRAME ?113, OPEN ?130, UNIT(NONE1) ?144, COMMA ?130,
    UNIT(NONE2) ?144, CLOSE /$DYNAM5(2);
#7# GENERAL-FRAME ?113, OPEN ?130, UNIT(NONE1) ?144, COMMA ?130,
    UNIT(NONE2) ?144, COMMA ?130, UNIT(NONE3) ?55, CLOSE
    /$DYNAM5(3).
DESCRIPTION(N):
#1#  /*SET0(N);
#2# DESCRIPTOR ?160, DIMENSION-LIST(N1) ?162, END-DESCR
    /$DESCR1(N,N1).
DIMENSION-LIST(N):
#1#  /*SET0(N);
#2# DIMENSION /*SET1(N);
#3# DIMENSION-LIST(N1) ?162, COMMA ?160, DIMENSION
    /*COUNT1(N,N1).
DIMENSION:
#1# TAG(NAME1) ?161, SUB ?163, INDEX-LIST(M1) ?164, BUS
    /$DESCR2(M1,NAME1);
#2# TAG(NAME1) ?161, OPEN-SLASH ?163, INDEX-LIST(M1) ?165,
    SLASH-CLOSE /$DESCR2(M1,NAME1).
INDEX-LIST(M):
#1# INDEX /*SET1(M);
#2# INDEX-LIST(M1) ?164, COMMA ?163, INDEX /*COUNT1(M,M1).
INDEX:
#1#  INDX /$OUTOPER(QIL2OP"PAIR",1);
#2#  INDX, COLON ?103 ,INDX /$OUTOPER(QIL2OP"PAIR",2).
INDX:
#1# TAG(NAME1) /$OUTPNTR(NAME1);
#2# DENOTATION(CONS1) /$OUTPNTR(CONS1).
# END OF GRAMMAR #  .
```

D ICF-macros

ICF (Intermediate Code File) is the language, produced by the code generator phase and processed by the code optimization phase. ICF-instructions are basically registerless machine code instructions.

ICF-macros allow the definition of operators in terms of machine-instructions rather than as an ALGOL 68 routine-text.

An ICF-macro consists of a header followed by a series of ICF-instructions.

```
ICF-macro ::= <ICF-header>
              {ICF-instruction}
              {ICF-instruction} .
```

An ICF-macro is used in an operation-definition of the form:

```
OP operator (parameter-list) result-mode =
PR inline
   <ICF-macro>
PR SKIP;
```

The form of the header is:

```
HEAD <count>,<length>, {S} {<id>}
```

where

<count> is an integer denoting the number of ICF-macro instructions in the macro,

<length> is an integer denoting the length of the result in words,

S is an optional parameter allowing the simultaneous definition of a number of operators with the same result mode, but different operand modes; one of the operands may be widened to the operand mode specified.

<id> is an optional parameter denoting an integer identifying the ICF-macro for later use.

An example of the use of the "S" parameter is the definition:

```
OP (COMPL, COMPL) COMPL + =
PR inline
   HEAD 2,2,S
1  fadd  /1,/3
2  fadd  /2,/4
PR SKIP;
```

defines five operators with the parameter plans:

```
(COMPL, COMPL) COMPL ,
(REAL , COMPL) COMPL ,
(INT  , COMPL) COMPL ,
(COMPL, REAL ) COMPL  and
(COMPL, INT  ) COMPL .
```

D.1 ICF-macro arguments.

An ICF-macro may have one to four arguments. The number and kind of these arguments are derived from the modes of the operands of the operator being defined.

Two kinds of arguments exist: value and address. Value arguments may be used as a value and address operands may be used as value or as address in ICF-instructions.

Each operand of the operator being defined gives rise to one or two arguments of the ICF-macro used in the definition. An operand of mode AMODE, where an object of mode AMODE occupies one or two words, gives raise to one or two value arguments.

An operand of mode BMODE, where an object of mode BMODE occupies more than two words gives rise to a single address argument.

Examples of operator operands and ICF-macro arguments are tabled below:

parameter pack	argument no	kind	representing
(INT a)	1	value	value of a
(INT a, REAL b)	1 2	value value	value of a value of r
(COMPL z, INT i)	1 2 3	value value value	RE z IM z value of i
(LONG COMPL zz)	1	address	address of zz
(REF LONG COMPL z)	1	value	value of z
([ ] INT i1)	1	address	address of descriptor of i1

D.2 ICF-instructions.

Syntax:

ICF-instruction ::=
    {<label>}{<resnr>}{ } <ICF-code> { } <operand>{,operand} .

<label>   ::= <letter>.
<resnr>   ::= <non-zero INT-denotation>.
<ICF-code> ::= <mnemonic>.
<operand> ::= <ICF-operand>.

Semantics.

<label> labels an ICF-instruction. The label may be used in subsequent ICF-instructions within the same macro to refer to the result of the labeled instruction.

<resnr> specifies the word of the result of the operator defined as the result of the instruction.

D.3 ICF-operands.

The number and kind of operands required in an ICF-instruction depend on the ICF-code used.

Five different types of operands are required for the different ICF-operators supported.

type	description
V	value operands representing a one word value resulting from a value argument of the macro, an address argument of the macro, a reference to a preceding ICF-instruction, an external name or a compiler variable.
A	address operand representing an address resulting from: an address argument of the ICF-macro or an external name.
O	address with offset representing an address computed at compile time resulting from an "OFFS" ICF-instruction
I	indexed address representing an address computed at run time by means of a "SUBS" instruction.
D	direct operand representing a fixed field of an instruction (e.g. the shift length in a shift instruction) resulting from a direct ICF-operand.

Syntactically ICF-operands defined by:

```
<ICF-operand> ::= *-<digit_sequence>|<label>|
                  =(-)ICF-denotation>=|$<ext-name>|
                  <digit-sequence>|R<digit-r>.
<digit-r>     ::= 1|2|3|4.
```

Semantics:

*-<digit-sequence>
, where <digit-sequence> denotes the value n, designates the result of the n-th preceding instruction in the macro.

<label> refers to the result of the instruction labeled <label>.

=(-)<ICF-denotation>=
denotes the constant value denoted by the ICF-denotation, complemented, if preceded by "-".

an ICF-denotation differs from an ALGOL 68 denotation in that it cannot begin with a LONG-symbol and that it uses bitswidth = 60 instead of 48.

$<ext-name>
denotes an external reference to the entity referred to by the external name.

<digit-sequence>
is called a direct operand and is used to specify constant fields of ICF-instructions.

R<digit-r>
designates a compile-time entity as follows:

digit-r	compile-time entity
1	THEN-label of the IF or WHILE construct
2	ELSE-label of the IF or WHILE construct
3	stack length of the innermost active routine.
4	current line number in the source text

"R1" and "R2" may be used only with conditional branch instructions.

D.4 ICF-code classification.

ICF-codes are categorized as follows:

A: arithmetic operations; four classes differing by number and type of operands and results.

B: branch; four ICF-instructions.

O: offset calculations for memory instructions.

M: memory instructions: load, replace, location.

P: pseudo instructions.

ICF-codes are divided in the following classes:

CAT.	CLASS	type of first operand	type of second operand	result type	description
A	RR/R	V	V	V	r := X1 OP X2
	R/R	V	-	V	r := OP X1
	R/RR	V	-	V	(r,b) := OP x
	RD/R	V	D	V	r := X1 OP D
B	JP	A, O	-	-	goto A1
	PCAL	A, O	-	-	call A
	IJP	A, O	V	-	goto A1+ V2
	CJP	A, O	V	-	IF V THEN goto A
O	OFFS	A	D	O	r := A1 + D2
	SUBS	A, O	V	I	r := A1 + V2
M	LOAD	A, O, I	-	V	r := MEM [V2]
	REPL	A, O, I	V	-	MEM[V1] := V2
	LOC	A, O, I	-	V	r := loc(MEM[A1])
P	SRV	V	D	-	X[D2] := V1
	DRV	D	-	V	r := X[D1]
	SELB	D	-	V	r := B-reg(D1]
	IS	D	-	V	r := X-reg[D1]
	LINE	D	D	-	line number
	EOS	-	-	-	end of sequence

Appendix E Garbage collector program.

```
BEGIN  #ALGOL 68 program#
#
 the following MODE indicants are used without definition:

 INT_dd,  where d is a digit, represents a dd-bit signed integer
 UINT_dd, where d is a digit, represents a dd-bit unsigned integer
 BIT      represents a one-bit BOOL
 LINK     represents an 18-bit address or in ALGOL 68 terms the UNION
          of all REF-modes.
#
MODE
 POINTER      = STRUCT (BIT sign, INT_11 exponent,
                        UINT_12 elt_size, BIT tracobj,
                        INT_10 dim, BIT compact,not_empty,
                        INT_6  not_used,
                        LINK address),
 DESCR_2      = STRUCT (INT_12 cdim, INT_48 span),
 TRIPLE_1     = STRUCT (INT lower),
 TRIPLE_2     = STRUCT (INT_36 width, INT_24 stride),
 LINK         = INT_18,
 WORD         = UNION (OUT_RANGE, DESCR_2,
                        TRIPLE_1, TRIPLE_2, HOLE_DES),
 OUT_RANGE    = UNION (POINTER, INDEF);
 HOLE_DES     = STRUCT (BIT_24 unused, LINK fwam1,lwa);


OP (POINTER) LINK ADDRESS = (POINTER p) LINK: address OF p;
OP (WORD) LINK ADDRESS = (WORD w) LINK :
  CASE w IN POINTER p: ADDRESS p OUT SKIP ESAC;


  [0:131071] WORD memory;

# garbage collector#
PROC garbage_collector = VOID:
BEGIN

[0:255]    INT ready_head, follow_head;
WORD  word,word2;
LINK  i, j, is, ij ,ic, lmin1;
POINTER pntr;
# Marking Phase#

INT   lwa,lmin1,reladr,blocknr,link;   #all   residing   in   the
                                       registers#
LINK ref_trac_chain;
PROC link_ready = VOID:            #link_ready calls are inline#
#
  link_ready chains a pointer into the appropriate chain of
  ready pointers
  the pointer is in pntr
  the address of the pointer is in ic
```

```
  the length of the object - 1 is in lmin1
#
  BEGIN
    lwa  :=  address  OF  pntr  +  lmin1;  #last_word_address  of
                                            object#
    reladr := lwa MOD 256;  #relative address of lwa in block#
    block_nr := lwa DIV 256;       #block_number#
    link := ready_head [reladr]    #header of chain#
    exponent OF pntr := block_nr; #store the block_number in
                                   the pointer#
    address OF pntr := link;
    memory[ic] = pntr;
    ready_head [reladr] := ic)     #new chain header#
  END;

# link_ready is implemented by means of the assembler macro RTAB,
  shown below.

RTAB    MACRO     BR
        SX6       X1+B2          LWA OF OBJECT REFERENCED
        BX3       -X7*X6         RELATIVE ADDRESS IN BLOCK
        AX6       L2BLK          BLOCK-NUMBER
        S_BR      X6             BLOCK-NR
        SA3       X3+RTAB        HEAD OF RCHAIN FOR THIS REL.ADR
        PX6       BR,X1
        BX6       X0*X6          CLEAR OBJECT ADDRESS
        IX6       X6+X3          INSERT LINK
        SA6       A1             RESTORE POINTER
        SX6       A1             ADDRESS OF POINTER
        SA6       A3             TO HEAD OF RCHAIN
        ENDM
#

PROC link_follow = VOID:
#
  link_follow chains a pointer into the appropriate chain of
  pointers to be visited again.
  The pointer is in word.
  The address of the pointer is in ic.
  The length of the object -1 is in lmin1.
  A specific hardware register (A2) is used to fetch the header
  of the follow_chain; this register is not used for any other
  purpose; thus its use sets a flag for the use of link_follow
  at no extra cost.
#
BEGIN
  lwa := address OF pntr + lmin1;
  reladr := lwa MOD 256;
  block_nr := lwa DIV 256;
  ref_trac_chain := follow_head [reladr];
  link := INT (ref_trac_chain);
  exponent OF pntr := block_nr;
  address OF pntr := link;
  memory[ic] := pntr;
  follow_head [reladr] := ic
```

```
END;

 # link_follow is implemented by means of the assembler macro
   FTAB, shown below. The A2 register holds ref_trac_chain.

FTAB    MACRO
        SX3       X1+B2            LWA OF OBJECT REFERENCED
        BX2       -X7*X3           RELATIVE ADDRESS IN BLOCK
        AX3       L2BLK            BLOCK-NUMBER.
        SB2       X3               BLOCK-NR
        SA2       X2+FTAB          HEAD OF FCAHIN FOR THIS REL
        PX6       B2,X1            BLOCK-NUMBER INTO POINTER
        BX6       X0*X6            CLEAR OBJECT ADDRESS
        IX6       X6+X2            INSERT LINK
        SA6       A1               RESTORE POINTER
        SX6       A1               ADDRESS OF POINTER
        SA6       A2               TO HEAD OF CHAIN
        ENDM
#
PROC folref = VOID:
#
  The procedure folref is the only out of line procedure in the
  garbage collector.
  Folref is called from scan_stack and scan_follow_chain.
  Folref traces all pointers in memory [i:j].
#
 (WHILE NOT i > j DO                    #scan the object#
   ic := i;
   word := memory[ic];
   i +:= 1;
   CASE word IN
   (POINTER pntr):
     BEGIN
tr_ref:
       IF sign OF pntr THEN             #a reference#
         lmin1 := size OF pntr;
         IF NOT tracobj OF pntr THEN    # a reference to a plain#
           link_ready                   # value#
         ELSEIF lmin1 > 0 THEN          #object to be traced and#
                                        # larger than one word#
           IF j > i THEN                #not tail of object,#
                                        #chain for later analysis#
             link_follow
           ELSE
             i := address OF pntr;      #last segment of object,#
             j := i + lmin1;            #set up new object to be#
                                        #scanned#
             link_ready;                #chain the pointer to#
                                        #the new object#
             folref                     #do tail recursion#
           FI    #descriptor of multiple of more than one word#
         ELSE                 #object is a single word#
           link_ready;
           ic := address OF pntr;
           word := memory[ic];
```

```
            CASE word IN
              (POINTER pntr_1):
               pntr := pntr_1; GOTO tr_ref; #recursion#
              OUT              SKIP
            ESAC
          FI  #reference #
        ELSE  #descriptor or ...#
          word2 := DESCR_DETAIL(memory[ic+1]);
          i +:= 2* dim OF pntr +1;
          lmin1 := span OF word2 - 1;
          IF lmin1 >= 0 THEN       #not a descriptor of  an  flat#
                                   #multiple#
            IF NOT tracobj OF pntr THEN      #multiple does not #
                                             #contain pointers#
              link_ready;
            ELSEIF cdim OF word2 > 0 OR j > i THEN
                #not compact and not last part of object#
              link_follow    #chain for later analysis#
            ELSE              #object is compact, needs#
                              #tracing and constitutes last#
                              # part of segment#
              i := address OF pntr;          #tail_recursion#
              j := i + lmin1                 #
            FI  #not empty#
          FI  #empty or not empty#
        FI   #reference or multiple descriptor#
      END  #pointer#
      OUT SKIP     #not a pointer #
    ESAC          #pointer or else...#
 OD;             #end of scan#

#
The actual assembler subroutine folref is shown below.

* SUBROUTINE FOLREF
* TRACES THE OBJECT OF A POINTER TO NON-PLAIN
* INPUT  B4  FWA OF OBJECT
*        B5  LWA OF OBJECT
* REGISTERS USED
* A0                  X0 MASK
* A1    WORKING       X1 CURRENT WORD      B1  - 1
* A2    ONLY IN FTAB  X2 PLAIN-BIT,SIZE    B2  SIZE,
* A3                  X3 SIZE,CDIM         B3  + 24
* A4                  X4                   B4  CURRENT ADDRESS
* A5                  X5                   B5  LWA OF OBJECT
* A6    WORKING       X6 WORKING           B6
* A7                  X7 MASK              B7
* HERE WITH A POINTER NON-PLAIN TO MORE THAN ONE WORD
FOLR1    GE        B5,B4,FOLR8   NOT TAIL OF OBJECT, CHAIN IT
* THIS NEW OBJECT IS SCANNED,AS IF IT WERE THE PARAM TO FOLREF
         SB4       X1            FWA OF NEW OBJECT
         SB5       X1+B2         LWA OF NEW OBJECT
FOLR2    RTAB      B2            CHAIN THE POINTER AS HANDLED
FOLR3    LT        B5,B4,FOLREF  IF OBJECT EXHAUSTED,EXIT
         SA1       B4            FETCH NEXT WORD


```

```
          SB4       B4-B1          COUNT
          IR        X1,FOLR3       PLAIN VALUE,GO NEXT WORD
          PL        X1,FOLR7       NOT A REF-WORD
* HERE WITH A REF-WORD
FOLR5     LX2       B3,X1          PLAINBIT=>BIT59,SIZE=>BIT17-0
          SB2       X2             SIZE
          PL        X2,FOLR2       IF REFERENCE TO PLAIN,CHAIN IT
* HERE WITH A POINTER NON-PLAIN
          LT        B0,B2,FOLR1
* HERE WITH A POINTER NON-PLAIN WITH A ONE-WORD OBJECT
          RTAB      B2             CHAIN THE POINTER
          SA1       X1             FETCH THE ONE OBJECT WORD.
          OR        X1,FOLR5       IF REFWORD
* HERE WITH REF TO ONE WORD PLAIN OBJECT, GO NEXT
          LT        B5,B4,FOLREF   IF OBJECT OF FOLREF EXHAUSTED
* HERE TO FETCH NEXT WORD
FOLR6     SA1       B4             FETCH NEXT WORD OF OBJECT
          SB4       B4-B1
          IR        X1,FOLR3       PLAIN VALUE,GO NEXT WORD
          NG        X1,FOLR5       REF-WORD
* HERE WITH A DESCRIPTOR OR A POSITIVE INFINITE PLAIN VALUE
FOLR7     SB2       X1
          GE        B0,B2,FOLR3
* HERE WITH A DESCRIPTOR, SIZE 2+2*NR OF DIMENSIONS
          SA3       A1-B1          VFD 12/CDIM,48ISPAN
          SB4       B4-B1          COUNT 1
          AX2       B3,X1          POSITION NR OF DIMENSIONS
          BX2       -X7*X2         ISOLATE DIM . NR OF DIMENSIONS
          SB4       X2+B4          COUNT DIM
          SB2       X3+B1          SPAN-1 .I.E LWA-FWA
          SB4       X2+B4          COUNT DIM
          LT        B2,B0,FOLR3    SPAN = 0, FLAT MULTIPLE
          LX2       B3,X1          PLAINBIT =>BIT59,SIZE=>BIT17-0
          PL        X2,FOLR2       IF REF TO PLAIN
          AX3       48             CDIM # 0 FOR NON-COMPACT SLICE
          ZR        X3,FOLR1       IF COMPACT,HANDLE AS REF-WORD
* HERE WITH A DESCRIPTOR OF A SLICE
* OR A POINTER TO AN OBJECT OF MORE THAN ONE WORD
* THIS POINTER CANNOT BE TRACED HERE AND
*  IS CHAINED FOR LATER TRACING
FOLR8     FTAB
          GE        B5,B4,FOLR6    NOT END OF OBJECT,GO NEXT WORD
* ENTRY / EXIT
FOLREF    DATA      0
          GE        B5,B4,FOLR6    OBJECT NOT EMPTY,GO NEXT WORD
          EQ        FOLREF
```

the scan_stack step traces the runtime stack
additionally it finds and interprets FORTRAN parameter list descriptors
the address of the first word of the stack is in is
the address of the last word of the stack is in js

The procedure tracref shown below is introduced here for descriptive purposes only. In the actual implementation all

```
 calls to tracref are replaced by inline code.
 the reference is in word
 the address of the reference is in ic
 the size of its object minus one is in lmin1
#
PROC tracref = VOID:
  (link_ready;              #the reference will not be#
                            #traced again#
  IF tracobj OF pntr THEN   #object requires tracing#
    IF lmin1 > 0 THEN       #contains more than one word#
      i := ic;
      j := i + lmin1;
      folref                # traces memory [i .. j]
    ELSE          #object is just one word#
                  # deref ; i.e. recursion turned into iteration#
      word := memory[address OF pntr];

      CASE word IN
        (POINTER pntr_1):
          IF sign OF pntr_1 THEN          #a reference#
            ic := address OF pntr;
            pntr := pntr_1;
            lmin1 := size OF pntr;
            tracref               #not a call, but a goto to the#
                                  #start-label of tracref#
          ELSE
            i := ic;
            j := i + lmin1;
            folref                # traces memory [i .. j]
          FI
        OUT SKIP            #the one word is plain; done#
      ESAC
    FI
  FI);  #end of tracref#

# code of the stack scan starts here#
  garlim : = top_of _memory;
  WHILE NOT is > js DO
    ic := is;
    word := memory[ic];
    is +:= 1;
    CASE word IN
     (POINTER pntr_1):
        pntr := pntr_1;
        lmin1 := size OF pntr;
        IF NOT sign OF pntr THEN            # not a reference#
          IF address OF pntr < 0 THEN       #ftn parameter-list#
            addr := address OF pntr + 131071;   #get its address#
            par_cnt := NRPAR pntr;          #number of parameters#
            WHILE
              par_adr := ADDRESS memory[addr+par_cnt];
              IF par_adr > top_of_stack AND
                 par_adr < garlim
              THEN
                garlim := par_adr;
```

```
            FI;
            par_cnt > 0
          DO
            par_cnt -:= 1;
          OD
        ELSE                              #descriptor of multiple#
          is +:= 2*dim OF pntr + 1; #adjust next address   #
                                          #for size of descriptor#
          word2 := DESCR_DETAIL(memory[ic+1]);
          lmin1 := span OF word2 - 1;
          IF NOT lmin1 < 0 THEN          # not flat#
            IF NOT tracobj OF pntr OR  # multiple does not#
                                          # contain any pointers#
              cdim OF word2 = 0          # multiple has no holes#
            THEN
              tracref                     #handle as a reference#
            ELSE
              link_follow             #chain for later analysis#
            FI
          FI                              # not flat#
        FI
      ELSE                                #a reference#
        tracref
      FI
    OUT SKIP                              #not a pointer #
  ESAC
OD;
# end of stack scan#

# scan of follow_chains #
INT chain_nr;
WHILE ref_trac_chain =/ NIL DO    #ref_trac_chain is used#
                                  #only in link_follow#
  ref_trac_chain = NIL;           #after each iteration#
                                  #ref_trac_chain = NIL#
                                  # means :no new entries in#
                                  #follow_chains#
  FOR chain_nr FROM 0 TO 255 DO
    link := follow_head[chain_nr];
    IF link =/ 0 THEN
      first := link;              #the first link#
      WHILE link =/ 0 DO
        pntr := POINTER (memory[link]);
        IF address OF pntr = 0 THEN  #the last link#

            #link the whole follow_chain in front of the#
            #corresponding ready_chain#
          pntr_1 := pntr;
          address OF pntr_1 := ready_head[chain_nr];
          ready_head[chain_nr] := first
        FI;
        block_nr := exponent OF pntr;
        lwa := 256*block_nr + chain_nr;
        lmin1 := size OF pntr;
        IF NOT sign OF pntr THEN        # not a reference#
```

```
          word2 := DESCR_DETAIL(memory[link+1]);
          lmin1 := span OF word2 - 1;
          cdimc := cdim OF word2;        # compactification#
                                         #metric#
        IF cdimc > 0 THEN                #not compact#
          dimc := dim OF pntr;
          IF cdimc < dimc THEN
            wid_str := TRIPLE_2(memory[link+1+2*cdimc]);
            lmin1 := width OF wid_str * stride OF wid_str-1;
                                  #size of compact segment#
          FI;
          FOR jt FROM cdim BY -1 TO 0 DO
            wid_str := TRIPLE_2 (memory [link + 1 + 2*jt]);
            count OF wid_str := width OF wid_str;
            memory [link + 1 + 2*jt] := wid_str;
          OD;
          jt := cdimc;
          base := lwa - lmin1;
op:
          i := base;             #address of active sub-slice#
          j := i + lmin1;        #last word address of same#
          folref;                #trace it#
          jt := cdimc;           #innermost triple#
          WHILE
            wid_str := TRIPLE_2 (memory [link + 1 + 2*jt]);
              #fetch triple jt  de-uniting coercion!!!#
            count OF widstr -:= 1; #step its count#
            memory [link + 1 + 2*jt] := wid_str; #save#
            base +:= stride OF wid_str;
              #adjust base for step in triple jt#
            IF count OF wid_str > 0 THEN GOTO loop FI;
              #count of triple jt not exhausted#
            base -:= width OF wid_str * stride OF wid_str;
              #undo ll base adjustments for triple jt#
            count OF wid_str := width OF wid_str;
              #reinitialize count of triple jt#
            memory [link + 1 + 2*jt]):= WORD (wid_str);
              #store initialized triple#
            jt -:= 1; #prepare to handle next outer triple#
            jt >= 0   #not all triples completed#
          DO SKIP OD;
      # reset count fields of all triples to zero#
          FOR jt FROM cdimc BY -1 TO 0 DO
            count OF TRIPLE_2 (memory [link + 1 + 2*jt]):= 0
          OD
        ELSE  #pointer does not refer to a slice with holes#
          i := lwa - lmin1;
          j := lwa;
          folref
        FI;
        link := address OF pntr;
      OD   # loop on chain not exhausted #
    FI   # chain processed IF head not zero #
  OD   # loop on chain_nr < 255 #
OD   # loop on ref_trac_chain =/ NIL #
```

```
#  scan_follow_chains end#


# sort pointers#
  ready_head[256] := -777; # mark end of chain table #
  chain_nr := 0;
  WHILE  # loop over chains #
    link := ready_head[chain_nr];
    NOT link < 0    # not marked as beyond table #
  DO # loop over chains #
    WHILE
      link > 0      # chain not empty #
    DO # scan one chain #
      pntr := POINTER(memory[link]);
      block_nr := exponent OF pntr;
      link_n := address OF pntr;
      address OF pntr := follow_head[block_nr];
      follow_head[block_nr] := link;
      exponent OF pntr := chain_nr;
      memory[link] := pntr;
      link := link_n
    OD # scan one chain #
  OD # loop over chains #
# sort pointers completed#

#adjust_pointer phase#

  HOLE_DES hole_descr;
  LINK lowm1, fwam1_hole, lwa_hole;
  chain_nr := fieldlg DIV 256; #highest numbered existing chain#
  WHILE
    chain_nr >= 0
  DO
    link := ready_head[chain_nr];
    ready_head[chain_nr] := 0; # for next garbage collection #
    chain_nr -:= 1;
    WHILE
      link > 0
    DO
      pntr := POINTER (memory[link]);
      reladr := exponent OF pntr;
      pntr_1 := pntr; address OF pntr_1 := 0;
      IF NOT sign OF pntr THEN  # multiple descriptor #
        word2 := DESCR_DETAIL(memory[link+1]);
        lmin1 := span OF word2 - 1
      ELSE                          # reference #
        lmin1 := size OF pntr
      FI;
      IF lwa < lowm1 THEN  # hole found #
        lwa_hole := adjblk + lowm1;
        fwam1_hole := adjblk + lwa;     # lwa current object #
        IF fwam1_hole < top_of_stack THEN
          GOTO no_adjust
        FI;
        IF lwa_hole < garlim THEN
```

```
          adjblk +:= fwam1_hole;
          adlblk -:= lwa_hole;
        FI;
        fwam1 OF hole_descr := fwam1_hole;
        lwa   OF hole_descr :=   lwa_hole;
        fwa := lwa - lmin1;
        memory[fwa_prev_hole] := hole_descr;
        fwa_prev_hole := fwam1_hole + 1;
        lowm1 := fwa - 1;
      ELSE
        fwa := lwa - lmin1;
        IF fwa - 1 < lowm1 THEN
          lowm1 := fwa - 1
        FI;
      FI;
      exponent OF pntr_1 := 1023; address OF pntr_1 +:= adj_blk;
      address OF pntr_1 +:= fwa;
      memory[link] := WORD(pntr_1);
      link := address OF pntr
    OD; # chain processed #
    adjblk -:= 256;
    lowm1 +:= 256;
  OD; # FOR chain_nr  #
  fwam1 OF hole_descr := 0;lwa OF hole_descr := lwa_hole;
  memory[fwa_prev_hole] := hole_descr;
  GOTO end_adjust;

no_adjust:
  fwam1 OF hole_descr := 0;lwa OF hole_descr := lwa_hole;
  memory[fwa_prev_hole] := hole_descr;
  adjblk := 256 * chain_nr;
  GOTO no_adj2; no_adj0:
  link := ready_head[chain_nr];
  IF NOT link = 0 THEN no_adj1:
    pntr := POINTER (memory[link]);
    reladr := exponent OF pntr;
    pntr_1 := pntr; address OF pntr_1 := 0;
    lmin1 := size OF pntr
    IF NOT sign OF pntr THEN  # multiple descriptor #
      word2 := DESCR_DETAIL(memory[link+1]);
      lmin1 := span OF word2 - 1
    FI; no_adj2:
    fwa := lwa - lmin1
    exponent OF pntr_1 := 1023; address OF pntr_1 +:= adj_blk;
    address OF pntr_1 +:= fwa;
    memory[link] := WORD(pntr_1);
    link := address OF pntr;
    IF link > 0 THEN
      GOTO no_adj1;
    FI
  FI;
  chain_nr -:= 1;
  adjblk -:= 256;
  IF chain_nr > 0 THEN GOTO no_adj0; end_adjust:
```

```
# adjust_pointers phase completed#

# compacting phase#
  hole_descr := first_hole;
  source := fwam OF hole_descr;
  WHILE NOT source < garlim
  DO
    hole_descr := memory [fwam OF hole_descr + 1];
    source := fwam OF hole_descr;
  OD;
  target := lwa OF hole_descr;
  WHILE source > 0
  DO
    hole_descr := memory [fwam OF hole_descr + 1];
    seg_length := source - lwa OF hole_descr;
    FOR i from 0 TO seglength -1 DO
      memory[target-i] := memory[source-i]
    OD
    target := target - seglength;
    source := fwam OF hole_descr;
  OD
#compacting phase completed#
END; #Garbage collector#
  user_program
END; #main program#
```

Stellingen behorende bij het proefschrift
"An Early Implementation of Revised ALGOL 68"
van J. Schlichting.

1. De taal ALGOL 68 is beinvloed door de beschrijvingsmethode. De operand modes van de LWB operator vormen een goed voorbeeld.

2. In ALGOL 68 is de plaatsing van declaraties niet aan dezelfde beperkingen onderhevig als in b.v. ALGOL 60 en Pascal, maar de consequenties hiervan zijn onvoldoende uitgewerkt. Verfijning van het scope begrip in ALGOL 68 zou leiden tot minder situaties, waarin tijdens executie getest moet worden of een bepaalde declaratie geëlaboreerd is.

3. Orthogonaal taalontwerp maakt het noodzakelijk het "goede" assenstelsel te kiezen.

4. Het in onderdelen opdelen van een taak voor dat men het gehele probleem doorziet leidt tot slechte implementaties.

5. Het werkelijke nadeel van one's complement representatie van integers is de onmogelijkheid modulo een macht van 2 te rekenen.

6. Inzicht in de door de compiler gegenereerde code bevordert het schrijven van efficiente programma's. Daarom is het wenselijk, dat programmeurs leren in assembler te programmeren.

7. De beschikbaarheid van goedkope snelle computers maakt efficientie van programmatuur niet overbodig.

8. Het ware wenselijk bij het ontwerpen van computers een grotere rol toe te kennen aan software overwegingen.

9. Vele Nederlandse termen voor woordsoorten zijn gevormd door deelwoordgewijze vertaling uit het Latijn, waardoor de betekenis niet op de in het gebruikelijke manier uit de betekenis van de samenstellende delen kan worden afgeleid.

10. Het GO-spel leent zich beter dan andere bordspelen tot oefeningen in strategie en tactiek.