PL/I

TECHNICAL REPORT

ULD VERSION III

TR 25.099
30 June 1969

INFORMAL INTRODUCTION TO THE ABSTRACT SYNTAX AND INTERPRETATION OF PL/I

K. ALBER
H. GOLDMANN
P. LAUER
P. LUCAS
P. OLIVA
H. STIGLEITNER
K. WALK
G. ZEISEL

IBM LABORATORY VIENNA

N O T E

This document is not an official PL/I Language Specification. For information concerning the official interpretation the reader is referred to the PL/I Language Specifications, Form No. Y33-6003-1.

IBM LABORATORY VIENNA,
Austria

INFORMAL INTRODUCTION TO THE ABSTRACT SYNTAX AND INTERPRETATION OF PL/I

by

K. ALBER
H. GOLDMANN
P. LAUER
P. LUCAS
P. OLIVA
H. STIGLEITNER
K. WALK
G. ZEISEL

ABSTRACT

This document represents an informal introduction to the formal definition of the abstract syntax and interpretation of PL/I. The intent of this document is to give sufficient information on the basis and structure of the formal definition so that questions of detail can be formulated and answered by consulting the formal definition.

Locator Terms for IBM Subject Index

PL/I
Formal Definition
Syntax, abstract
Semantics
21 PROGRAMMING

TR 25.099

30 June 1969

PREFACE

This document is an updated version of:

/1/ LUCAS, P., ALBER, K., BANDAT, K., BEKIC, H., OLIVA, P., WALK, K., ZEISEL, G.: Informal Introduction to the Abstract Syntax and Interpretation of PL/I.
IBM Laboratory Vienna, Techn. Report TR 25.083, 28 June 1968.

It is part of a series of documents (ULD Version III) presenting the formal definition of syntax and semantics of PL/I:

/2/ FLECK, M.: Formal Definition of the PL/I Compile Time Facilities (ULD Version III).
IBM Laboratory Vienna, Techn. Report TR 25.095, 30 June 1969.

/3/ URSCHLER, G.: Concrete Syntax of PL/I (ULD Version III).
IBM Laboratory Vienna, Techn. Report TR 25.096, 30 June 1969.

/4/ URSCHLER, G.: Translation of PL/I into Abstract Syntax (ULD Version III).
IBM Laboratory Vienna, Techn. Report TR 25.097, 30 June 1969.

/5/ WALK, K., ALBER, K., FLECK, M., GOLDMANN, H., LAUER, P., MOSER, E., OLIVA, P., STIGLEITNER, H., ZEISEL, G.: Abstract Syntax and Interpretation of PL/I (ULD Version III).
IBM Laboratory Vienna, Techn. Report TR 25.098, 30 April 1969

/6/ ALBER, K., GOLDMANN, H., LAUER, P., LUCAS, P., OLIVA, P., STIGLEITNER, H., WALK, K., ZEISEL, G.: Informal Introduction to the Abstract Syntax and Interpretation of PL/I (ULD Version III).
IBM Laboratory Vienna, Techn. Report TR 25.099, 30 June 1969.

The method and notation used in these documents are essentially taken over from the first version of a formal definition of PL/I issued by the Vienna Laboratory:

/7/ PL/I Definition Group of the Vienna Laboratory: Formal Definition of PL/I.
IBM Laboratory Vienna, Techn. Report TR 25.071, 30 December 1966

/8/ ALBER, K.: Syntactical Description of PL/I Text and its Translation into Abstract Normal Form.
IBM Laboratory Vienna, Techn. Report TR 25.074, 14 April 1967.

An outline of the method is given in:

/9/ LUCAS, P., LAUER, P., STIGLEITNER, H.: Method and Notation for the Formal Definition of Programming Languages.
IBM Laboratory Vienna, Techn. Report TR 25.087, 28 June 1968.

This document also contains the appropriate references to the relevant literature. The basic ideas and their application to PL/I have been made available through several workshops on the formal definition of PL/I, and presentations and publications inside and outside IBM. The method is demonstrated by application to an appropriately tailored subset of PL/I in:

/10/ LUCAS, P., WALK, K.: On the Formal Description of PL/I.
To be published in Annual Review in Automatic Programming - Vol.6.
Pergamon Press, New York 1969.

The language defined in the present version is PL/I as specified in the PL/I Language Specifications, Form No. Y33-6003-1, with the addition of attention handling, input stream and string scanning, and file variables.

The present document will be made subject to validation by the PL/I Language Department, Hursley.

CONTRIBUTION TO THE DOCUMENT

Project manager: K. WALK

Authors: Authors of the present version are listed in the following with their main contributions indicated by chapters. It should be stated that credit must be given also to those authors of previous versions who have not been engaged in producing the present document. The contributions of these authors are listed in the respective documents of the previous versions.

K. ALBER	2., 7., 8., 9.
H. GOLDMANN	13.
P. LAUER	4.1, 10.3
P. LUCAS	5.
P. OLIVA	4.3, 12.
H. STIGLEITNER	14.
K. WALK	1., 3., 4.2, 6., 10.1, 10.2
G. ZEISEL	11.

Proofreading: P. LAUER

Production: This document was prepared by means of automated text-processing systems. TEXT 360 was used for processing the prose parts. The formatting, indexing, cross-referencing, and updating of formula texts was handled by means of FORMULA 360.

FORMULA 360 is a syntax-controlled formula processing system which was developed in the Vienna Laboratory especially to facilitate the production and maintenance of PL/I Formal Definition documents. The achievements of K.F. KOCH in the overall design and implementation of FORMULA 360 are acknowledged in particular. Essential components of the system are due to G. URSCHLER (syntactical decomposition of formulas) and E. MOSER (formula input checker). H. Hoja and G. Zeisel contributed to the clarification and formulation of the required formatting processes.

Coordination: F. Schwarzenberger, M. Stadler

Technical control: K.F. Koch, E. Moser, M. Stadler

Data transcription: Miss W. Schatzl, Mrs. H. Deim, and sub-contractors

System support: H. Chladek, G. Lehmayer

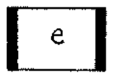
SCOPE OF THE DOCUMENT

This document is an informal introduction to the formal definition of the abstract syntax and interpretation of PL/I /5/. The intent of this document is to give sufficient information on the basis and structure of the formal definition so that questions of detail can be formulated and answered by consulting the formal document. The central part of this introduction starts with an outline of the main syntactic structures of abstract programs with attached notes as to the relation of abstract programs to their concrete representation (chapter 2). A brief summary of the major state components of the PL/I machine is given in chapter 3. The chapter on storage and data outlines the objects which can be manipulated by a PL/I program (e.g., values, value representations, datasets,...) and their treatment by the formal definition. A summary follows of the entities which can be declared in a PL/I program and their formal equivalents. The computation of the PL/I machine is discussed in chapter 6. Chapters 7 through 14 explain the basic behaviour of the PL/I machine in interpreting the major components of a program, whereby the instruction definitions of the formal definition and the control cycle of the PL/I machine are replaced, so to speak, by plain English sentences sometimes augmented by flow charts.

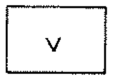
This document is neither an introduction to the notation used nor to the method applied in the formal definition (for method and notation see /9/ or /10/). The terminolgy and style of the explanation assumes familiarity with PL/I and with the methodology of the formal definition. This document, therefore, does not represent a self-contained introduction to PL/I and is only intended to be used in connection with the formal document /5/. It does not cover the entire range of the formal definition of PL/I.

NOTATION

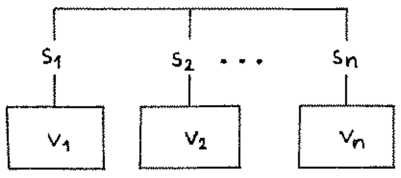
In general, abstract objects are represented in a two-dimensional form. The following conventions are used:



elementary object. The box contains either a variable whose name indicates the type of the object or the object itself;



composite object whose structure is not further specified. The box contains either a variable whose name indicates the type of the object or a concrete representation of the object itself;



composite object where $s_1, s_2, \dots, s_n$ are selectors and $v_1, v_2, \dots, v_n$ are the immediate components.

CONTENTS

1. THE OVERALL STRUCTURE OF THE FORMAL DEFINITION OF PL/I

The block diagram (Fig. 1.1) shows the process which is taken as the basis of the formal definition of PL/I.

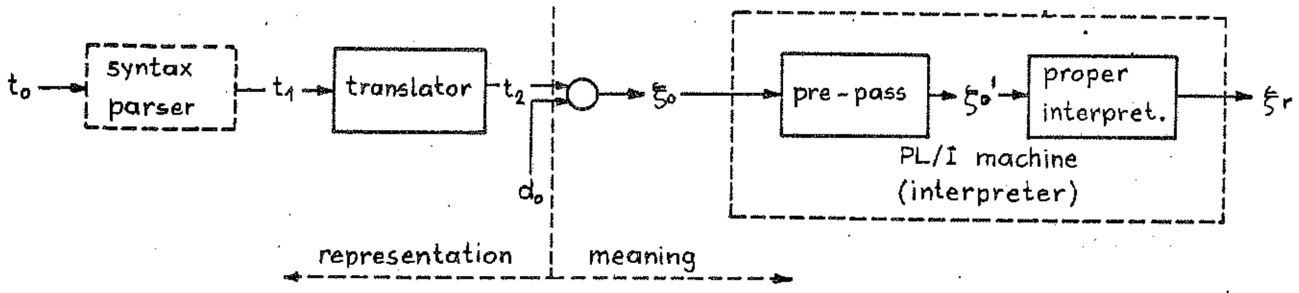


Fig. 1.1 Structure of the formal definition

The input to the entire process is the concrete program (concrete text) t_0, which is a PL/I program represented as a character string, and certain initial data sets d_0. The set of concrete programs considered by the formal definition is defined by a set of syntactic rules (in extended Backus Normal Form) called the concrete syntax.

No specific process has been specified for the syntax parser (therefore shown in dotted lines) whose result (the parsing tree t_1) is implied by the concrete syntax.

The translator has been specified by a function which maps the parsing tree t_1 into the abstract program (abstract text) t_2. The task of the translator is to keep the structure of the parsing tree where this structure is significant, to transform the program into some standard form where the structure is not significant (e.g., the translator collects all declarations spread over one block into one component of the block) and to remove some notational conventions (e.g. partially qualified names are fully qualified by the translator). The result of the translator is an abstract object as described in /9/, i.e., a tree with named branches and elementary objects at the terminal nodes, which exhibits the essential structure of the PL/I program. All abstract programs considered by the further process are defined by the abstract syntax. The set of programs specified by the abstract syntax is a superset of the set of programs which can be produced by the translator for the parsing tree considered.

The rest of the interpretation is defined by the PL/I machine whose initial state ξ_0 is produced essentially from the abstract program t_2 and the initial data-sets d_0. The machine may be considered to run through a sequence of states, called the computation, while it interprets a program until an end state is reached (if ever). In principle, the interpreter as specified by the formal definition allows (and this is its task) the generation of a computation for a given PL/I program and given data sets. More precisely, because the interpreter is not fully determined, it allows the generation of a set of possible computations. The interpreter is specified as a function which yields for any state the set of possible successor states. For the following reasons it may be the case that the computation actually cannot be produced by the formal definition of the interpreter:

(1) because the evaluation of an implementation-defined (and therefore unknown) function is necessary

(2) because a partially defined function has to be applied to an argument for which the function does not have a defined value.

Any state of the PL/I machine is an abstract object as described in /9/, i.e., the same formal tools can be applied to the abstract program and the states of the PL/I machine. The set of all states which the PL/I machine can possibly assume for any given abstract program and any data sets is contained in the set of states defined by the abstract syntax of the states. The abstract syntax of the states exhibits the essential structure of the states of the PL/I machine.

The process defined by the interpreter falls into two major parts, the prepass and the proper interpretation.

The prepass accomplishes the following tasks:

(1) allocation and initialization of static variables

(2) null allocation of controlled variables

(3) linkage of the scope of the external declarations

(4) insertion of appropriate information into the declarations occurring in the program to establish the necessary linkage between the declarations and the entries made in the state of the PL/I machine during the prepass (see (1), (2), (3) above).

The intermediate state ξ_0' contains then the abstract program modified according to (4).

Finally the proper interpretation interprets the prepassed program according to the meaning of the individual statements.

The abstract syntax of PL/I may be taken as the center of the formal definition in the sense that the process to the left of the dotted line in Fig. 1.1 deals with a special representation of PL/I as a character string and the process to the right deals with the meaning of PL/I.

Only the abstract syntax and the interpreter are considered in this document.

2. STRUCTURE OF ABSTRACT PROGRAMS

Corresponding chapter of /5/:

2. Abstract syntax of program

The following abbreviations are used in this chapter:

ADD	addition
aggr	aggregate attribute
AL	aligned
ap	argument part
AUTO	automatic
BIN	binary
cond	condition
const	constant
CONV	conversion
CTL	controlled
da	data attribute
DEC	decimal
decl	declaration
dens	density
descr	descriptor
DIV	division
elem	element
eva	evaluated aggregate attribute
expr	expression
EXT	external
FIX	fixed
FLT	floating point
id	identifier
init	initial
INT	internal
lbd	lower bound
MULT	multiplication
op	operand
opr	operator
param,PARAM	parameter
prec	precision
prop-st	proper statement
ptr	pointer
qual	qualification
ref	reference
ret-type	return type
scale-f	scale factor
sl	subscript list
st,stmt	statement
st-loc	statement location
stg-cl	storage class
ubd	upper bound
UNAL	unaligned
v	value

This chapter describes the overall structure of abstract programs, i.e., the abstract syntax of PL/I. Where necessary the correspondence with the concrete syntax is given. An abstract program, referred to as <u>program</u> throughout this document, is an abstract object as described in /5/.

<u>2.1 BLOCK STRUCTURE</u>

Corresponding sections of /5/:

2.1 Program, procedure body

2.3.1 Block, group

A PL/I program constitutes a nested structure of <u>blocks</u>: A program itself may be considered as a block and certain of its subcomponents are blocks. Since programs are objects, their structure implies that the blocks contained in a program are themselves tree structures: If b_1 and b_2 are two different blocks in a program then: either b_1 contains b_2, or b_2 contains b_1, or b_1 and b_2 are two distinct components in a common containing block b.

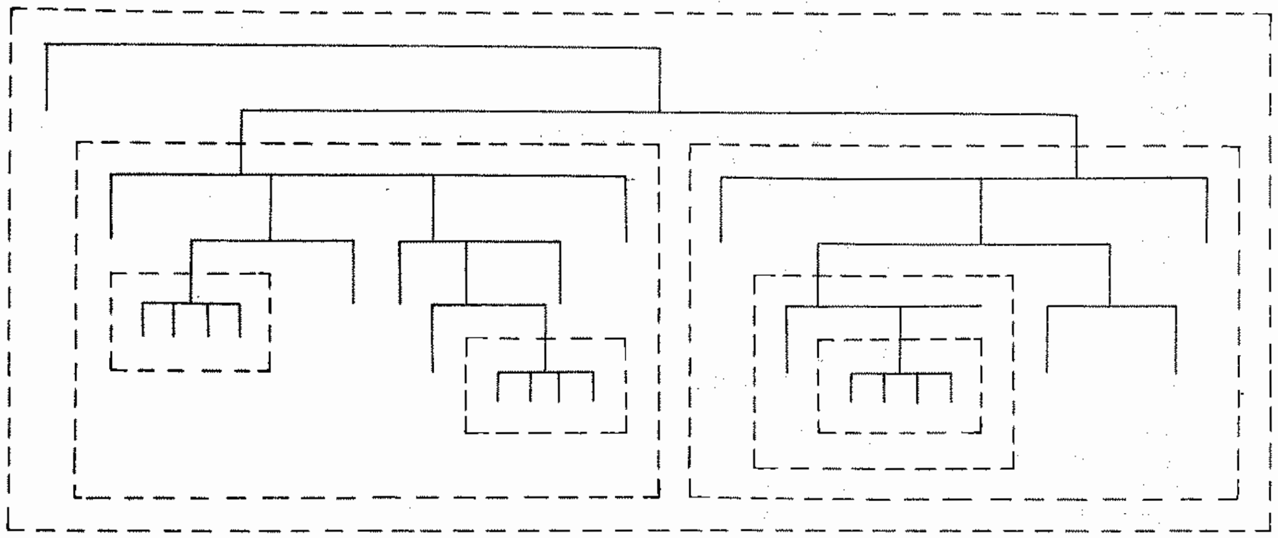


Fig. 2.1 Object structure and block structure of a program (the dotted boxes indicate blocks).

During the interpretation of a program, the interpretation of a block establishes a so-called block activation which introduces:

(1) new meaning of identifiers,

(2) a new enabling status of conditions,

(3) a new level in the nested structure of statements,

(4) a new optimizing status.

Accordingly a block generally consists of the following five comoponents:

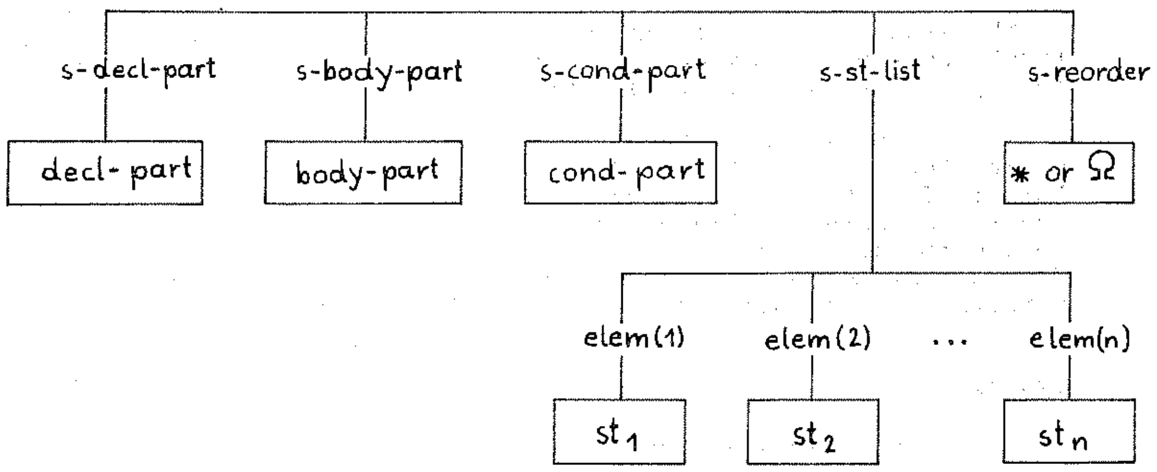


Fig. 2.2 General structure of a block

(1) A declaration part collecting all declarations local to the block (whether they are explicit, contextual or implicit in the concrete program); the structure of a declaration part is described in 2.2.

(2) A procedure body part collecting all procedure bodies local to the block; the structure of a procedure body part and the relation between an entry declaration and the corresponding procedure body is described in 2.2.2.

(3) A condition prefix part consisting of two lists of conditions, namely those conditions to be enabled and those to be disabled for the block.

(4) A list of statements, which constitutes the main part of the block. This list contains only the "executable" statements at the outermost level in the block in their consecutive order, ignoring all declarative information (declarations, procedures, format sentences, entry points) contained intermixed in the concrete program text. A discussion of the term "statement" and a description of the structure of statements is given in 2.3.

(5) A flag indicating whether it is a reorder block or not. This is used for optimizing purposes only (cf. 14).

There are essentially two types of blocks: begin blocks and procedure bodies. As a third type, also a complete program itself may be considered as a block.

2.1.1 PROGRAM

A program is an incomplete block consisting only of a declaration part and a procedure body part. The conditions to be enabled by default are fixed in the initial state of the PL/I machine. Instead of a statement list a single call statement or function reference is interpreted, which is specified apart from the program (the "initial call" of the "main procedure"). It is not a reorder block.

The declaration part consists of the declarations of all entry identifiers of all external procedures of the program. The procedure body part consists of all external procedure bodies.

It is assumed that the program contains all external procedure bodies needed for its execution and the corresponding entry declarations. That is: If for an external entry declaration the concrete program does not contain a corresponding procedure body, an external procedure body has to be incorporated from outside the concrete program, e.g. from a library. In the abstract program it is assumed that this process has been performed by an implementation defined function used by the translator.

<u>Example</u>:

The following concrete program:

```
A:B:PROC ...          }
    DCL E ENTRY EXT;  }
    ...               }
  C:ENTRY ...         }  body_A
    ...               }
    END A;            }
  D:PROC ...          }
    ...               }  body_D
    END D;            }
```

is translated into:

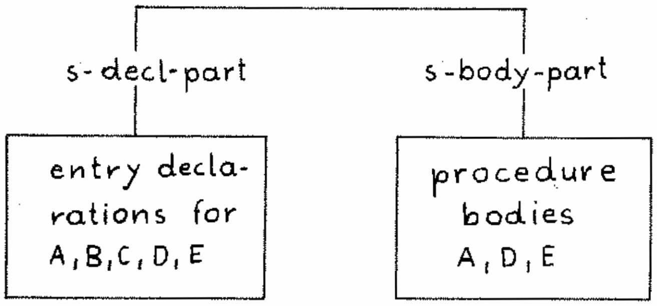


Fig. 2.3 Example of a program

2.1.2 BEGIN BLOCK

A begin block occurs anywhere in the context of a proper statement (cf. 2.3). In fact, it is a proper statement. As such it is activated when the normal flow of control comes to the execution of this proper statement, and after its termination the flow of control continues normally. The structure of a begin block is exactly the general structure of blocks as described above.

Example:

The concrete begin block

```
(CONV,NOSIZE):BEGIN REORDER;

              DCL A ...;

              statement-1;

              P:PROC ... END P;

              statement-2;

              DCL X ..., Y ...;

              L:statement-3;

              END;
```

is translated into:

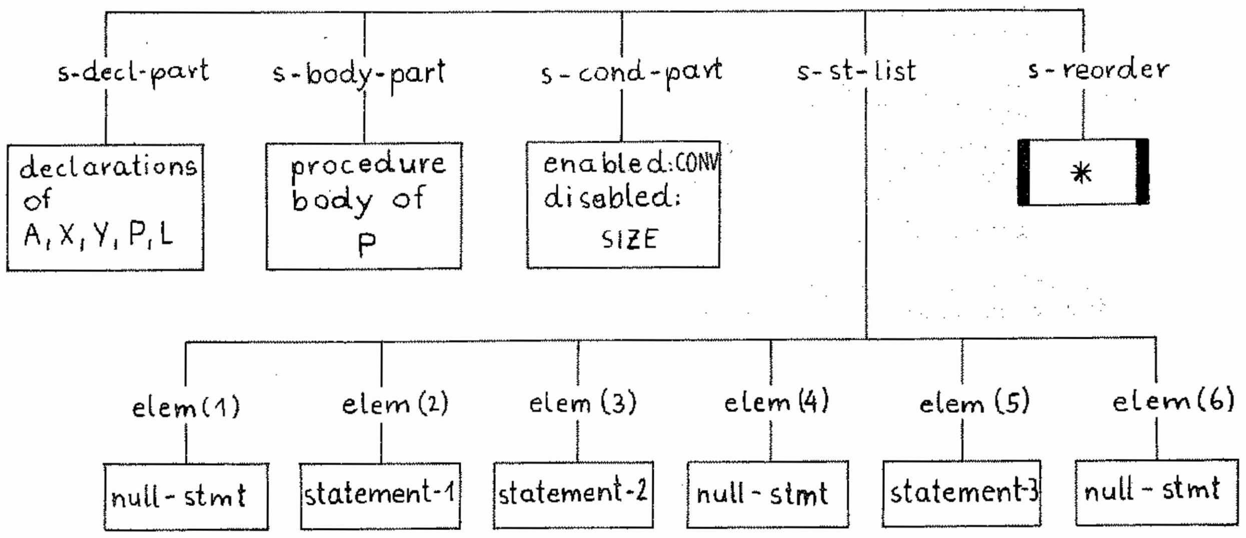


Fig. 2.4 Example of a begin block

Note: Cf. 2.3 for the treatment of the concrete DCL's and END's in the abstract program.

2.1.3 PROCEDURE BODY

A procedure body occurs as an immediate component of the procedure body part of a block. It is activated by a call statement or function reference by means of an <u>entry identifier</u>. The same procedure body may be activated by means of different entry identifiers.

A procedure body contains, besides the five general components of any block as described above, a sixth component, the <u>entry part</u>. It contains for each entry identifier associated with the procedure body an individual component, called <u>entry point</u>. (Note: There is no difference between the "main entry" and a "secondary entry" in an abstract program; different entry identifiers occurring at the same entry point in a concrete program have their individual entry points in the translated abstract program).

The entry point of an entry identifier consists of three components:

(1) The <u>statement location</u>. It is an index list localizing relatively to the statement list of the procedure body that statement by which the interpretation of the statement list is to be started if the procedure body is activated by this entry identifier. This component is constructed by the translator using the position of the entry point in the concrete text. The localization of a statement within a statement list by means of an index is described in 9.4.

(2) The <u>parameter list</u>. It is the list of those parameter identifiers to whom arguments are passed when the procedure body is activated by means of this entry identifier. (Note: For convenience of the interpreter the identifiers are not themselves elements of the parameter list; they are appended by the selector s-id to the list elements; cf. 8.2.1).

(3) The <u>return type</u>. It specifies the data attribute (and density) to which the function value is to be converted before return if the procedure body is activated by a function reference by this entry identifier. The return type is constructed by the translator from the returns attribute explicitly specified in the concrete text or inserted by default.

Additionally, a procedure body may contain as seventh component a flag indicating that the procedure may be invoked recursively.

Example:

The concrete body:

```
(CONV,NOSIZE):Q:I:PROC(X,Y) REORDER RECURSIVE;
                             DCL A ...;
                             statement-1;
                           P:PROC ... END P;
                           R:ENTRY(X) FIXED;
                             statement-2;
                             DCL X ..., Y ...;
                             statement-3;
                             END Q;
```

is translated into:

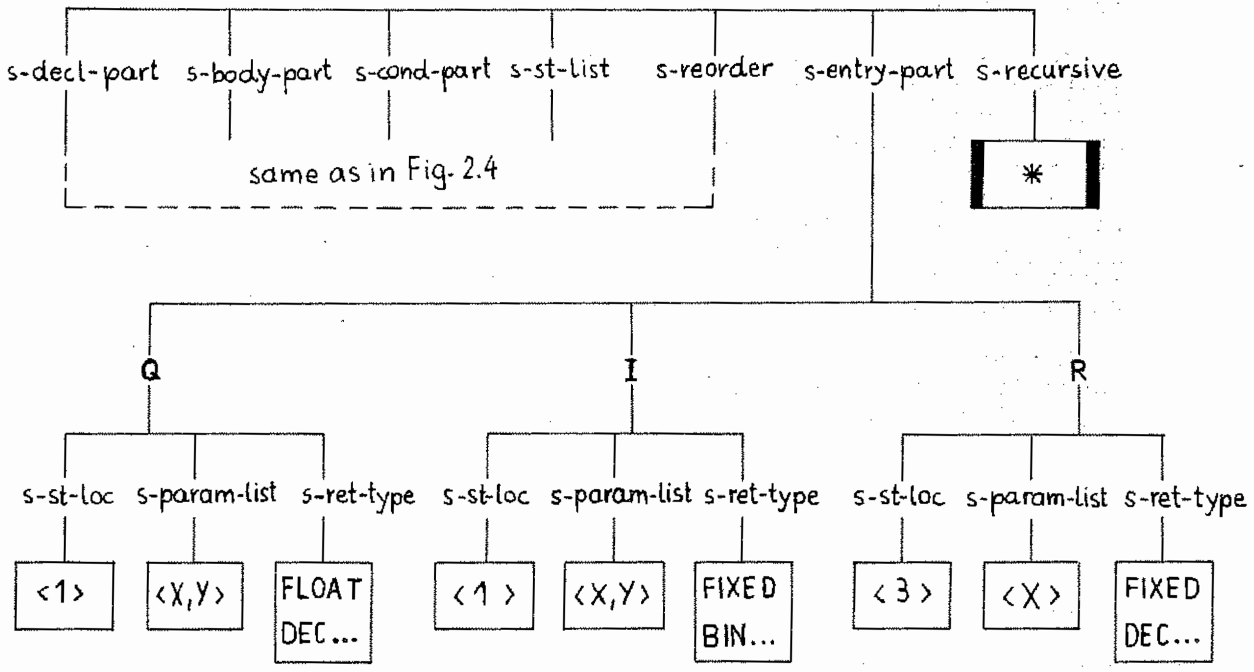


Fig. 2.5 Example of a procedure body

2.2 DECLARATIONS

Corresponding sections of /5/:

2.2. Declarations

2.5.1 Evaluated aggregate attributes

In the declaration part of a block all declarative information is collected which is local to the block, except the bodies of the local procedures which are collected into the separate body part. This declarative information is all information valid for the whole block independently from its location within the concrete program text of the block. Each identifier declared local to the block, whether its declaration in the concrete program is explicit, contextual or implicit, has a declaration in the declaration part, with the following exception. For a structure declaration only the major structure identifier, the main identifier of the structure, has a declaration and not the identifiers of the components of structures. The declarations are complete in the sense that all attributes implied by default statements or by the default rules of the language from the concretely specified attributes are inserted by the translator. Therefore the default statements of a concrete program do not appear in the abstract program anymore.

To each identifier of the concrete program corresponds uniquely an abstract identifier which is an elementary object satisfying the predicate is-id. The transformation from the character string representing an identifier in the concrete program to its corresponding abstract identifier is performed by the function mk-id (cf. chapter 1 of /5/). In the following the term identifier denotes such an abstract identifier while the identifiers of the concrete program are denoted as concrete identifiers where necessary. Nevertheless, in figures the abstract identifiers are represented by the corresponding concrete representations (e.g. A is written instead of mk-id(A)).

The structure of a declaration part is the following: Each declared identifier serves as selector selecting its declaration from the declaration part.

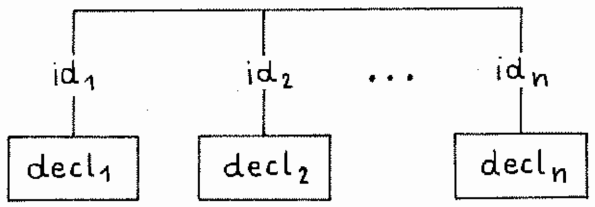


Fig. 2.6 Declaration part

This structure of a declaration part provides easy access to an individual declaration through the declared identifier itself; any other structure would require a more complicated device for accessing an individual declaration.

Example:

The declaration part of the block given as example in 2.1.2 has the following structure:

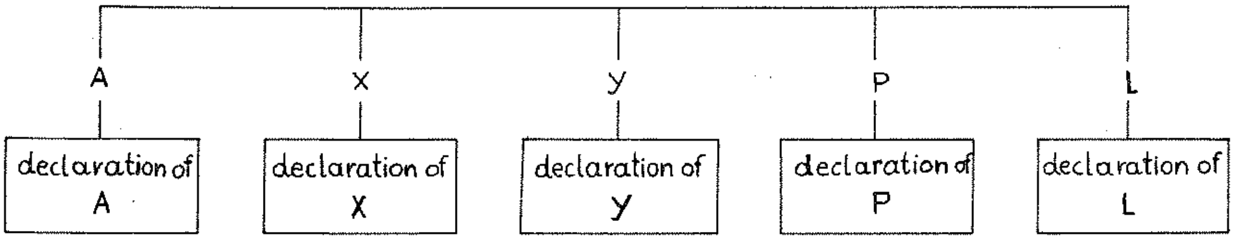


Fig. 2.7 Example of a declaration part

Each individual declaration is an object, whose structure depends essentially on the type of the declaration. There are the following eleven types of declarations:

(1) Proper variables. Their declarations are described in some detail in 2.2.1.

(2) Defined variables. Their declarations consist of: a reference to the base variable, the aggregate attribute of the defined variable (as described in 2.2.1) and possibly an expression for the position in the case of overlay defining.

(3) Based variables. Their declarations consist of: the aggregate attribute of the based variable (cf. 2.2.1) and possibly a reference to the implied pointer.

(4) Entry constants. Their declarations and their correspondence to the procedure bodies are described in 2.2.2.

(5) File constants. Their declarations consist of: the set of file attributes (as far as explicitly declared in the concrete program), the scope (INT or EXT) and the implementation dependent environment attribute.

(6) Statement label constants. Each statement label constant has in the abstract program as its declaration an index list (list of integers and truth values) which localizes the labeled statement relative to the statement of the containing block. This localization is described in 9.4. The index list is constructed by the translator from the position of the labeled statement within the concrete text.

(7) Format label constants. Format sentences, which in the concrete program have the syntactical form of statements, are declarative information and occur in the abstract program as declarations of their labels and not as statements. These declarations consist of the format list, the condition prefix part and an identifier uniquely identifying the format sentence (the first label of the format sentence is chosen by the translator): A format sentence with two labels leads to two declarations with the same identifier, while two identical format sentences lead to two declarations with different identifiers.

(8) Generic identifiers. Their declarations are lists of generic family members. A generic family member consists of the entry reference to be selected and the list of possibly incomplete parameter descriptors.

(9) Builtin functions. For each builtin function used, whether it is declared explicitly with the attribute BUILTIN in the concrete text or not, in the abstract program there is a declaration which is just the elementary object BUILTIN.

(10) Programmer named conditions. Their declaration is the elementary object COND.

(11) Attentions. Their declarations contain only an implementation dependent environment attribute.

2.2.1 PROPER VARIABLES

The term proper variable denotes variables of any storage class (static, automatic and controlled) and parameters. It does not include defined or based variables. It denotes only "level-one variables", i.e., arrays, structures and scalars which are not themselves components of arrays or structures.

The declaration of a proper variable consists of three or four components:

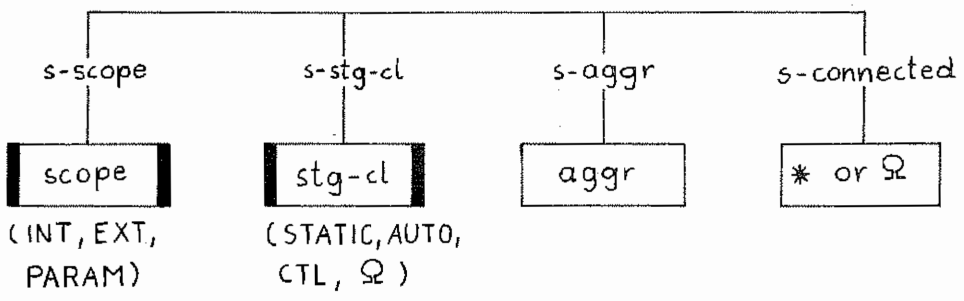


Fig. 2.8 Declaration of a proper variable

These components are:

(1) The scope attribute INT, EXT or PARAM. In an abstract program parameter declarations have their own scope attribute PARAM. This is useful since the distinction between internal, external and parameter declarations is often needed.

(2) The storage class attribute STATIC, AUTO or CTL. Non-controlled parameter declarations have the null object Ω as storage class component.

(3) The aggregate attribute as described below in more detail.

(4) Non-controlled parameters may have a flag indicating that only connected arguments may be passed to them.

The aggregate attribute (aggr) of a proper variable declaration (and of a defined or based variable declaration as well) reflects the complete structuring of an aggregate (array, structure, scalar): Since, during interpretation, it appears easiest to handle data aggregates by recursively defined functions or instructions level by level, the structuring of data attributes is decribed level by level:

(1) Arrays. A multi-dimensional array is decomposed into a nested sequence of one-dimensional arrays; e.g. a two-dimensional array of scalars is handled as a one-dimensional array, whose elements are themselves one-dimensional arrays of scalars. An array of structures is naturally handled in the same way with the only difference that its base elements are described as structures. Hence, an abstract program describes only one-dimensional array aggregates; the elements may be arrays, structures or scalars. Array aggregate attributes consist of three components: An expression or asterisk denoting the lower bound (if missing in the concrete program, the constant 1 is inserted by the translator), an expression or asterisk denoting the upper bound and the aggregate attribute of the elements of the array:

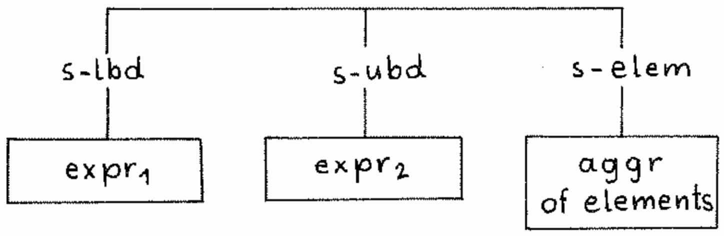


Fig. 2.9 Aggregate attribute of an array

A refer option occurring as array bound (or string length or area size) in the aggregate attribute of a based variable is translated into an object consisting of the initializing expression and an identifier list which is the fully qualified name of the referenced structure component without the main identifier.

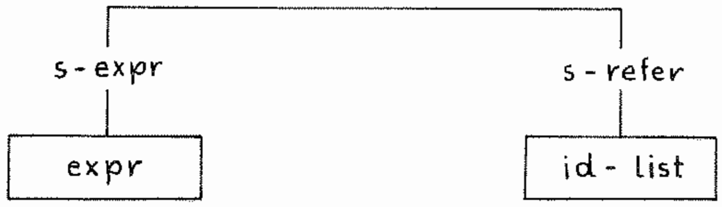


Fig. 2.10 Refer option

(2) Structures. Like arrays, structures are described recursively. A structure is analogous to a one-dimensional array, whose elements may have any aggregate attribute: array, structure, or scalar. The difference is that all elements of an array have the same description, while for a structure all elements (called successors) have to be described separately and to be listed in their given order. Furthermore each successor has to be named by

an identifier (which is used in references to the successor by a qualified name). So, the description of a successor of a structure consists of two components, identifier and aggregate attribute, and a complete structure description has the form:

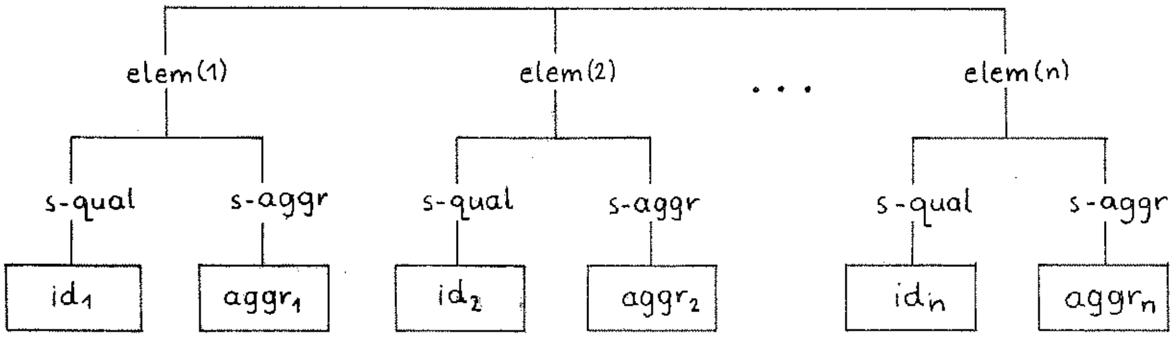


Fig. 2.11 Aggregate attribute of a structure

Notes: (a) The LIKE attribute occurring in a concrete program is removed by the translator and replaced by the complete aggregate description.

(b) The main identifier of a structure occurs as selector of the complete declaration in the declaration part, while the successor identifiers are s-qual components in the aggregate attribute.

(3) <u>Scalars</u>. By this recursive description of the structuring of data aggregates, one finally comes down to the scalar components. The aggregate attribute of a scalar consists generally of three components:

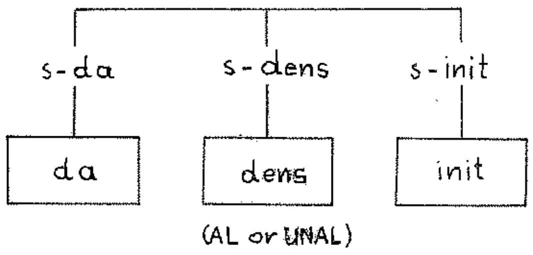


Fig. 2.12 Aggregate attribute of a scalar

(a) The <u>data attribute</u>. It is an object describing the properties of the individual types of data: mode, base, scale, precision and scale factor for arithmetic data; string base, an expression or asterisk denoting the length and a flag distinguishing varying or fixed length for string data; parameter descrptor list, return type and reducible flag for entry data (cf. 2.2.2); etc.

(b) The <u>density</u>(ALIGNED or UNALIGNED). This attribute is a property of scalars, though it may be written in a concrete program also for aggregates. The translator resolves the language rules for the inheritance of the density attribute from aggregates to their components.

(c) The initial attribute (if applicable). It specifies the initialization of the scalar or, if it is a component of an array at any level, the initialization of all corresponding components of that array. There are three types of initial attributes: Nested lists of expressions with replication factors, or call statements, or lists of "special initial elements". The latter are produced by the translator in case subscripted statement labels occur in the concrete program text representing initial values for an array of label variables. In this case, for the subscripted statement labels label constant declarations are produced with newly created identifiers; for the array of label variables a list of special initial elements is produced, each consisting of the subscript list specifying which element of the array is to be initialized and the newly created identifier as initial value. The same applies for subscripted entry names as initial values for arrays of entry variables.

Example:

The concrete program text

```
DCL 1 Z(5,N:N+M), CTL ,
      2 A(*) INIT(0,0),
      2 B,
        3 C BIT(M+1),
        3 L LABEL;
...
L(1,3):statement-1;
...
L(2,5):statement-2;
...
```

leads to the following declaration part:

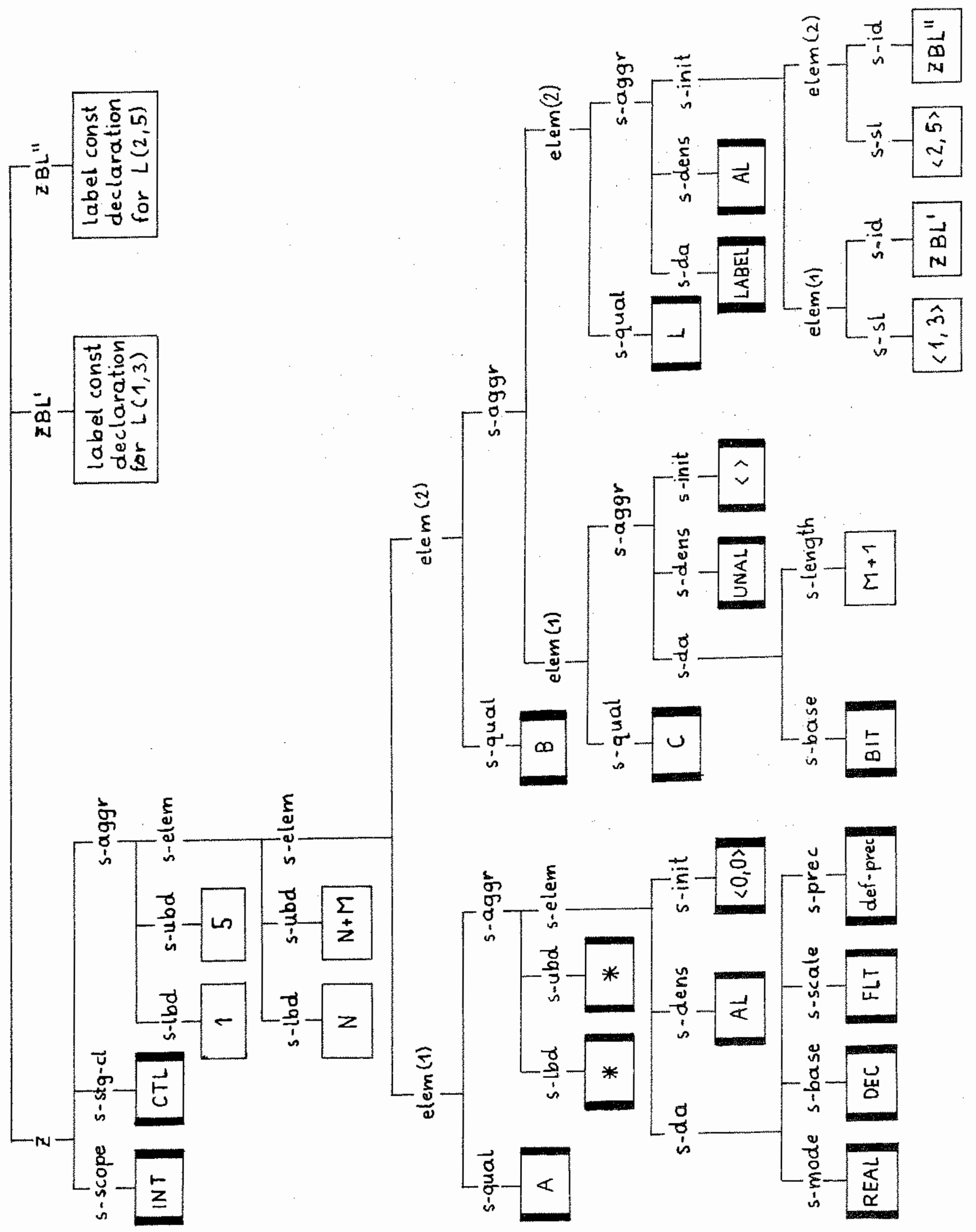


Fig. 2.13 Example of a proper variable declaration

Often during interpretation evaluated aggregate attributes (eva) are needed, especially for storage mapping. These are objects, produced during the interpretation, which have a very similar structure to the aggregate attributes described above, with the following differences:

(1) The expressions denoting extents (lower and upper bounds of arrays,string lengths, area sizes) are evaluated and replaced by their (integer) values. Even if there are only integer constants as extents, they have to be evaluated, i.e., replaced by their values, since a constant is a more complicated object than just its value (cf. 2.5.2).

(2) The identifiers of successors, which are irrelevant for storage mapping are deleted.

(3) The initial attributes are removed.

(4) For several types of data (entry, label, offset), information which is irrelevant for storage mapping is removed. Their data attributes are replaced by standard ones, namely elementary objects (ENTRY, LABEL, OFFSET).

These evaluated aggregate attributes contain exactly the information necessary for storage mapping of the described aggregate.

Example:

The evaluated aggregate attribute of the declaration in the previous example (Fig. 2.12) has the following structure (assuming that N has the value 2 and M the value 4).

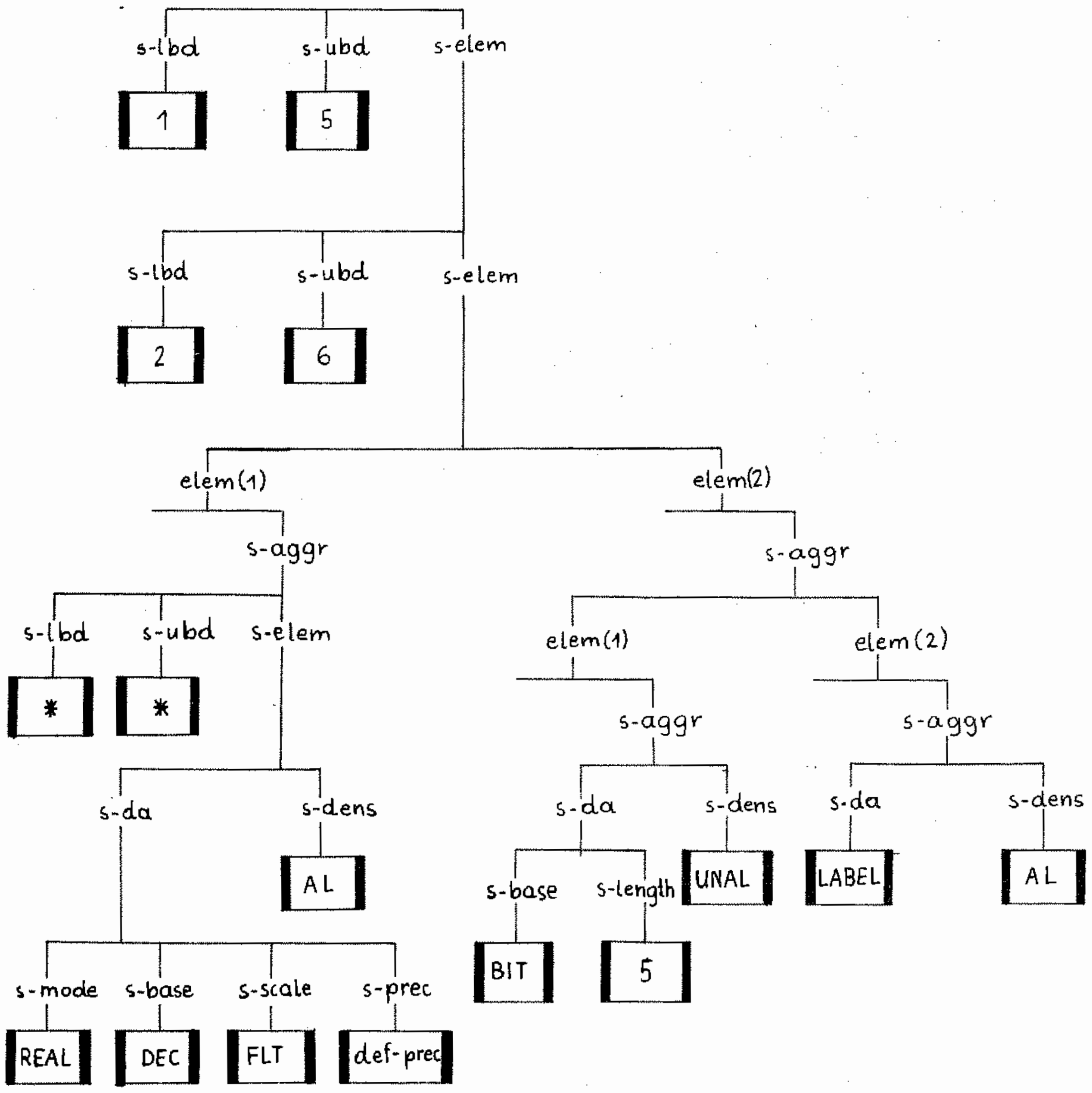


Fig. 2.14 Example of evaluated aggregate attribute.
Note, that in this figure digits in the boxes denote values, while in the previous ones they denote the corresponding constants.

2.2.2 ENTRY DECLARATIONS

Both declarations of entry constants and of entry variables contain the following three components:

(1) The parameter descriptor list to be used on invokation for conversion and testing of arguments. A parameter descriptor has the form of an incomplete parameter declaration: It consists of: a storage class (CTL or Ω), an incomplete aggregate attribute (no qualifying identifiers, no initial attributes and only integer values or asterisks as array bounds, string lengths or area sizes), and possibly a flag denoting a connected parameter.

The parameter descriptor list is produced by the translator from the parameter descriptor list of the entry attribute in the concrete program. If no parameter descriptor list is specified (or a single parameter descriptor is missing) in the concrete program, the translator takes the information from the parameter declarations in the corresponding procedure body in those cases where a procedure body is available. This is the case for internal entry constants and also for external entry constants in the declaration part of the program itself. In the other cases (external entry constants in inner blocks of the program and entry variables) missing descriptor lists or missing single parameter descriptors are substituted by an *.

Example:

The concrete program text:

```
  DCL P ENTRY(,FLOAT(10) CONNECTED);
  ...
P:PROC(X,Y);
  ...
  DCL X BIT(N) ALIGNED CTL;
  ...
  END P;
```

leads to the following parameter descriptor list in the declaration of P:

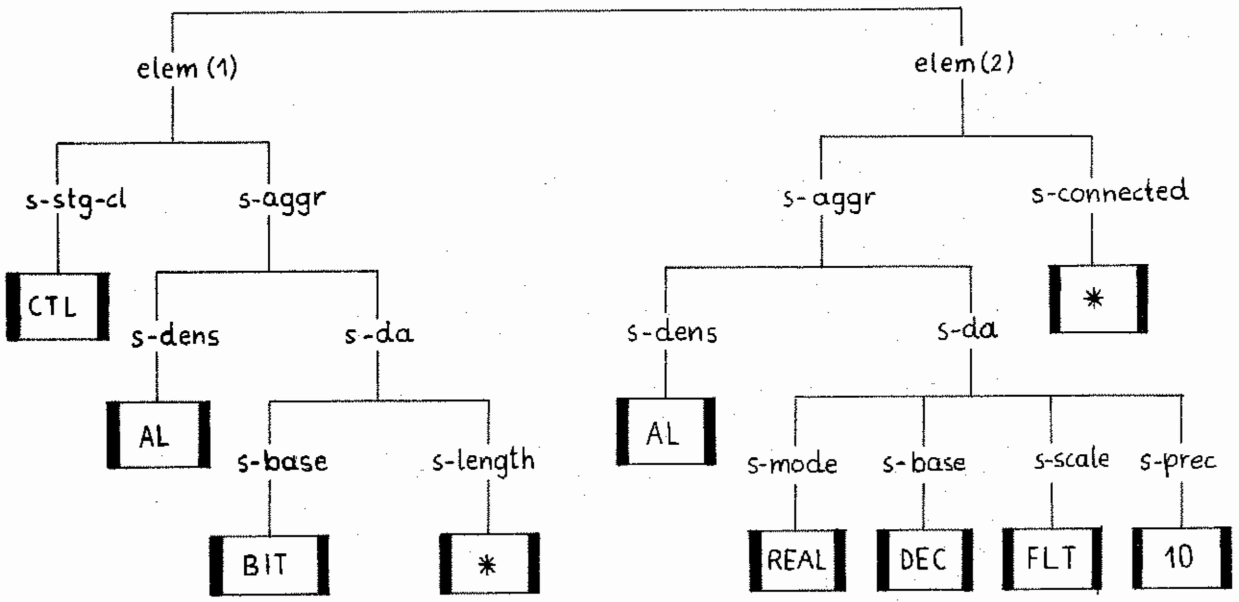


Fig. 2.15 Example of a parameter descriptor list

(2) The <u>return type</u> specifying the data attribute and density of the returned function value in case of a function reference. The return type is produced by the translator from the returns attribute of the entry declaration in the concrete program. If no returns attribute is specified the translator takes the information from the returns attribute in the corresponding procedure body in those cases where a procedure body is available (as described above for the parameter descriptors). Otherwise the return type is produced by the default rules of the language.

(3) A flag denoting whether the corresponding procedure is reducible or irreducible.

Note: These three components of an entry declaration are used when the declared entry is invoked. It is the responsibility of the programmer that they fit the invoked entry point of a procedure body. If not, it is an error. In particular, it is possible to assign any entry constant to any entry variable, irrespective of whether the declarations fit together or not.

In addition to these three components, which are common to declarations of entry constants and to data attributes of entry variables, the declarations of entry constants have the following components:

(4) The scope attribute: INT or EXT. For internal entry constants there exists a procedure body with a corresponding entry point in the procedure body part of the same block in which the entry constant is declared. For external entry constants there exists a procedure body with a corresponding entry point in the procedure body part of the program.

(5) Declarations of internal entry constants (and of external entry constants in the declaration part of the program itself) have a s-body component which is an identifier giving the link to the corresponding procedure body, as described below.

As mentioned above (in 2.1) all procedures which are local to a block b are combined in one of its components, the procedure body part s-body-part(b). Each procedure body bd is an immediate component of the procedure body part, selected by an identifier id, i.e., bd=id•s-body-part(b), (the translator takes the first entry identifier of the procedure in the concrete program for this purpose, though this choice is completely irrelevant for the meaning of the program). With a procedure body bd, a number of entry identifiers are associated; each of them, id', gives access to an entry point ep as described in 2.1.3: ep=id'•s-entry-part(bd). For each of these entry identifiers id' there is an individual entry constant declaration decl in the declaration part of the containg block b, i.e., decl=id'•s-decl-part(b). These different entry constant declaraions belonging to the same procedure body have as s-body component that identifier id, by which the procedure body is selected from the procedure body part: s-body(decl)=id. Thus, each internal entry constant declaration gives access, via its s-body component, to a corresponding procedure body; conversely each procedure body gives access, via the selectors of its entry points, to a number of entry constant declarations.

Example:

The concrete program text

```
A:BEGIN;
  ...
  P:Q:PROC ... ;END P;
  ...
  R:PROC ... ;S:ENTRY ... END R;
  ...
  END A;
```

leads in the above program to the following relevant components of the block A:

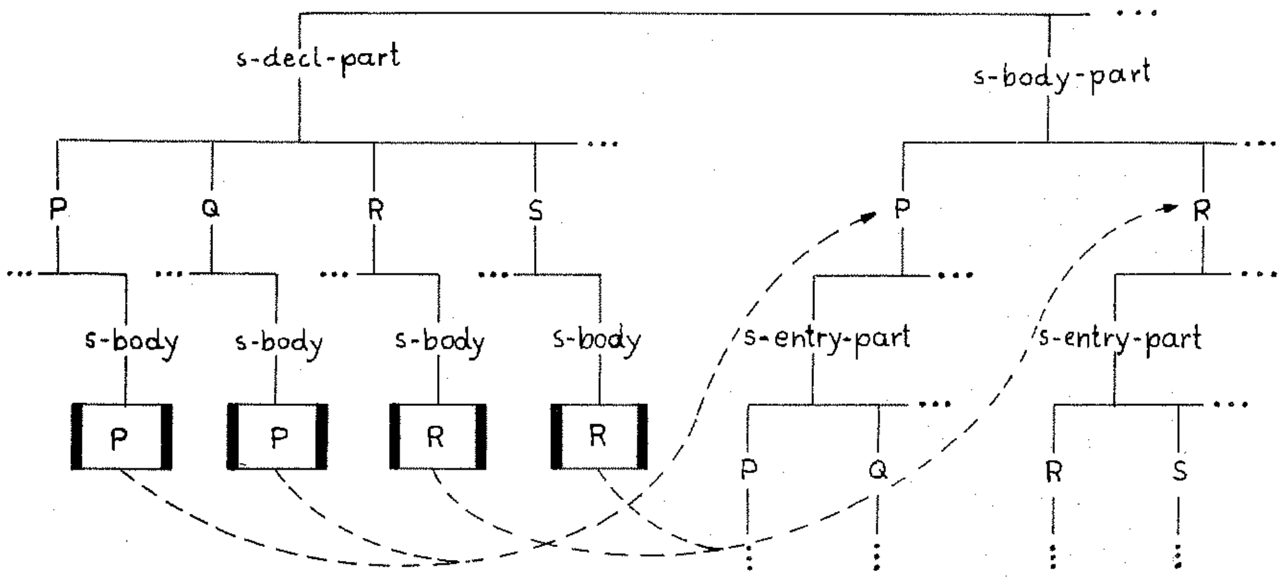


Fig. 2.16 Relation between internal entry declaration and procedure body

For external entry constants, the procedure bodies are found in a different way. For each external entry constant declared in any block in the program, there is an entry constant declaration of the same identifier in the declaration part of the program itself. This declaration is connected with a procedure body in the procedure body part of the program in the same way as described above. So, an external entry constant declaration has a s-body component only if it occurs in the declaration part of the program itself.

<u>Example</u>:

The program

```
A:B:PROC ...
    DCL B;
    ...
    BEGIN;
       DCL B ENTRY EXT;
       ...
       END B;
    ...
    END A;
```

leads to the following situation:

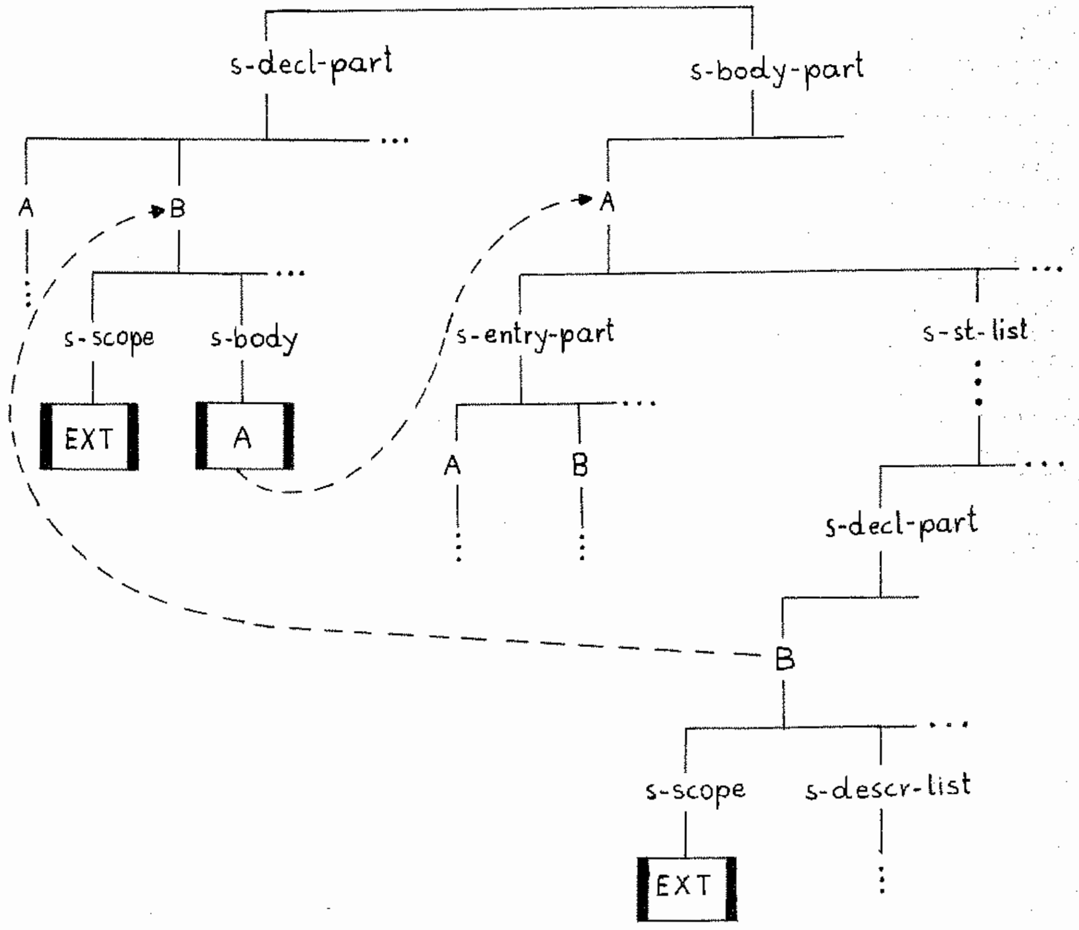


Fig. 2.17 Relation between external entry declaration and procedure body.

The association of entry variables (including parameters) with procedure bodies is performed dynamically during the interpretation by assignment.

2.3 STATEMENTS

Corresponding section of /5/:

2.3 Statements

Throughout the formal definition of PL/I the term statement denotes a logically complete unit of program text to be executed during the sequential flow of control at the point given by its position within the program. The term includes: the simple statements (e.g., assignment statement, goto statement, null statement), the if-, on- and access statement, the different types of do-groups and the begin block. It does not include: declarations, procedure bodies, entry points, format sentences and incomplete clauses as, e.g., BEGIN; or DO I= 1 TO N; or IF 2 > 1 THEN or END; etc. So, the term "statement" does not denote the units syntactically delimited by semicolons in the concrete program, but logical units that may appear anywhere "in a statement context", e.g., as THEN alternative of an if statement, and that may in some way be executed independently from other program parts.

The main part of a block (begin block or procedure body) is a list of statements to be executed one after the other in their given order (cf. 2.1). Some of these statements may themselves contain statements (namely the if-, on- and access statement) or even lists of statements (namely the different groups and the begin block). Since these contained statements principally may be any type of statements and thus may themselves contain statements, the statement list of a block may be not just a linear sequence of statements but a rather complicated structure of nested statements.

Each statement has primarily the same structure: It consists of the following three components:

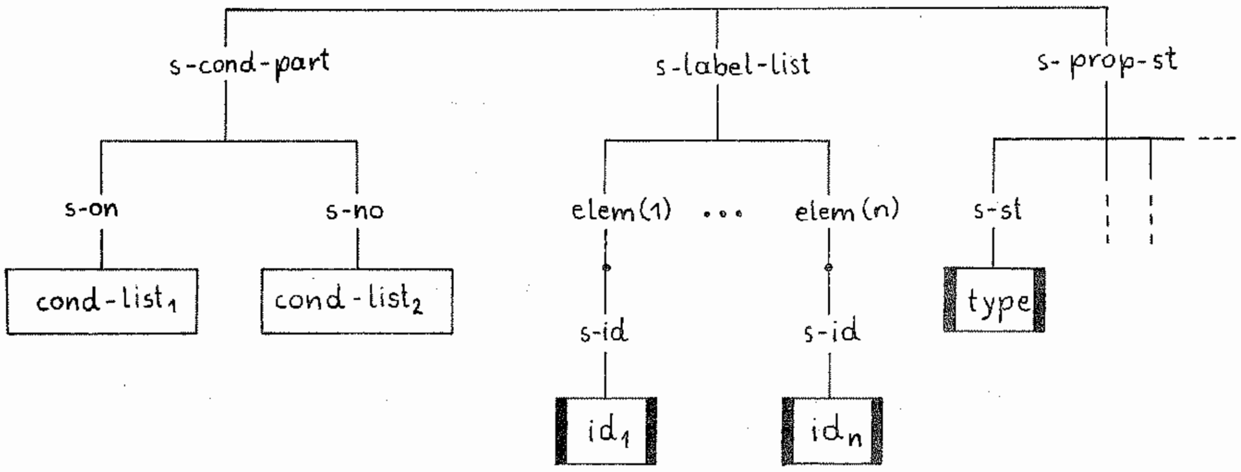


Fig. 2.18 General structure of a statement

(1) a <u>condition part</u>, consisting of the list of conditions to be enabled and of the list of conditions to be disabled for the statement.

(2) a <u>label list</u>. For convenience of the interpreter the label identifiers are not themselves elements of the list; they are appended by the selector s-id to the list elements. The label list is used only for the purpose of raising the check condition. Information about a statement label for the purpose of the goto statement is taken from the label declaration (cf. 2.2, 9.6).

(3) the <u>proper statement</u>. There are 35 different types of proper statements (including begin block and group). The structure of a proper statement depends very much on its individual type. In

most cases it is a nearly one-to-one mapping of the syntactical structure in the concrete program.

Principally it would be possible to recognize the type of a proper statement from its structure (e.g., there is no other statement with a left part s-lp, and a right part, s-rp than the assignment statement). But there are some pairs of statement types (e.g., open and close, get and put) which, at least in special cases, may not be distinguishable by the structure of the statement alone. Therefore all proper statements, except begin block and group, have a component, s-st, which is an elementary object denoting the statement type.

Since the structure of the statements is a one-to-one mapping of their syntactic form in the concrete syntax, they are not enumerated and described in detail here (the reader may get all relevant information from section 2.3 of /5/). The following are only some special additional remarks, mentioning some deviations between the abstract and concrete syntax.

Begin block. In the structure of a program, the begin blocks play a double role. On the one hand, they are proper statements, and thus they occur in the structure of statements where any other proper statement, e.g., an assignment statement, might occur. On the other hand, as described in 2.1, they impose (together with the procedure bodies) the block structure upon the program. It should be noted, that the condition prefixes occurring in front of a begin block in a concrete program are translated into the block condition part inside the begin block (cf. 2.1) and not, as for all other statements, into the statement condition part beside it (that condition part consists of two empty lists). This is because the condition prefixes of a begin block have a different semantical meaning from those of other statements.

Group and statement list. There are two essentially different "do-groups" in a concrete program: those with iteration specification and those without it. Only those with iteration specification are called groups throughout the formal definition. Those without iteration specification are considered just as statement list, parenthesized in a concrete program by the parentheses DO; and END;. Thus a proper statement may itself be just a statement list in the abstract program.

If-statement. The if-statement has always two alternative statements. If there is no else alternative specified in the concrete program, the translator inserts a null statement.

Null statement. A null statement in an abstract program may result not only from a concrete null statement, but also from a missing else alternative of an if-statement (as mentioned above) or from a concrete declare or default sentence or end clause. This is because all declarative information in a declare or default sentence is translated into the declaration part, but possible labels have a semantical meaning and must hold their position within the structure of statements.

Allocate statement. Deviating from the structure in the concrete syntax, the information about one data aggregate to be allocated is collected into one component and structured similarly to a proper variable declaration, in particular its aggregate attribute (cf. 2.2.1). Thereby the qualifying substructure identifiers, which are redundant, are omitted.

2.4 EXPRESSIONS

Corresponding section of /5/:

2.4 Expressions

Expressions are decomposed by the translator into (possibly nested) "elementary expressions". There are five different forms of elementary expressions:

(1) an _infix expression_, consisting of two operand expressions and an infix operator (which is an elementary object),

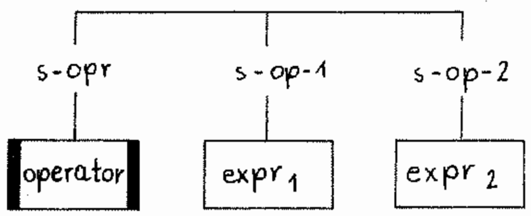


Fig. 2.19 Infix expression

(2) a _prefix expression_, consisting of an operand expression and a prefix operator (which is an elementary object),

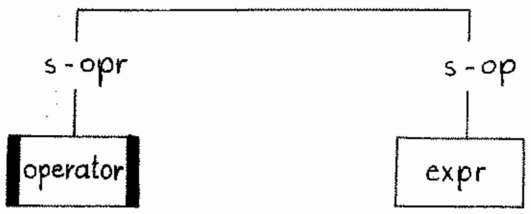


Fig. 2.20 Prefix expression

(3) a _parenthesized expression_, consisting of an operand expression only,

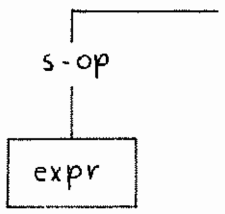


Fig. 2.21 Parenthesized expression

(4) a <u>reference</u> (described below, cf. 2.4.1),

(5) a <u>constant</u> (described below, cf. 2.4.2).

This decomposition reflects the operations to be performed one after another when evaluating the expression. Moreover, it resolves the precedence rules of the operators of the language since this structure determines uniquely the operands for each operator.

In principle, the parentheses of a concrete program could be eliminated by the translator producing structured objects as already described. But since in the language there is one case (argument passing) where parentheses have more than syntactical meaning, the <u>parenthesized expressions</u> are left in the abstract program in the form of an object having only one component, namely the translation of the concrete expression contained in the parentheses.

<u>Example</u>:

The concrete expression

- A * B + (X + Y) / C

is translated into the object:

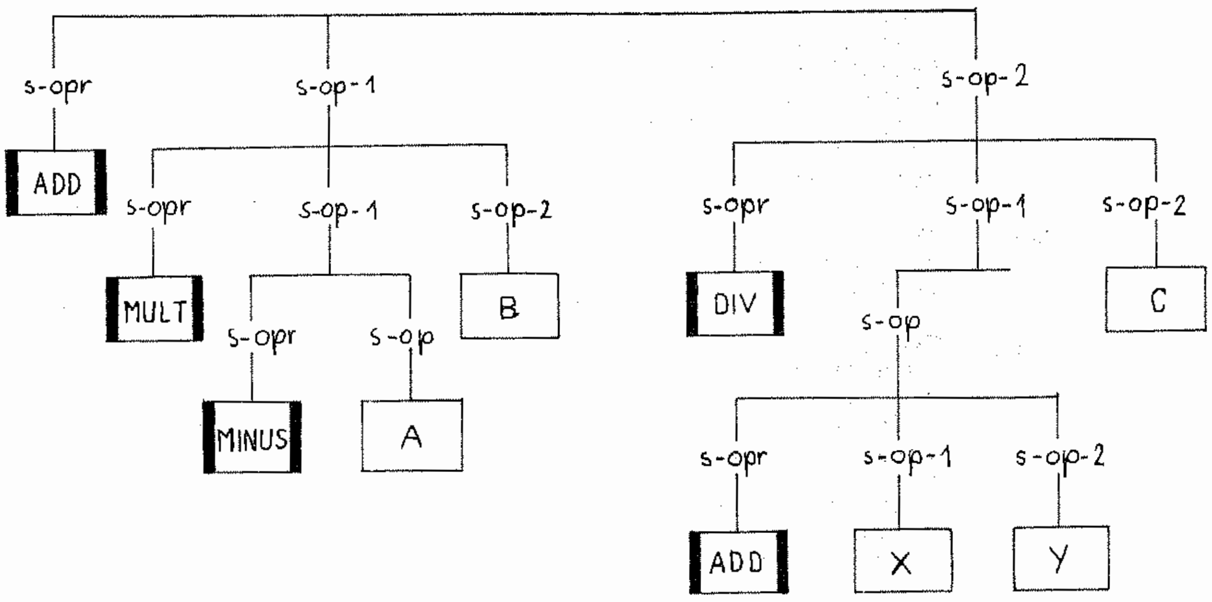


Fig. 2.22 Example of an expression (the boxes for A,B,... represent references as described below)

The final components of an expression are references and constants.

2.4.1 REFERENCES

A reference may refer to a variable (proper, defined or based variable) or to an entry, statement label, format label or file constant. It is an object consisting of the following four components:

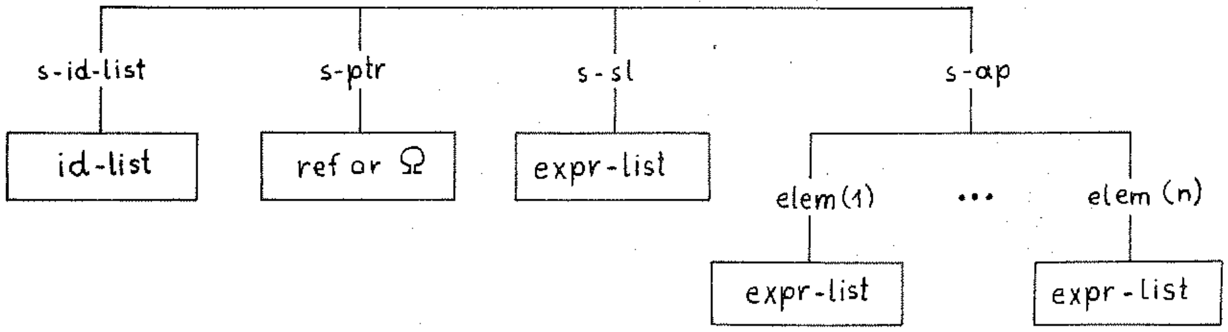


Fig. 2.23 Structure of a reference

(1) The identifier list. In the case of a reference to a constant it consists of only one identifier, that of the constant. In the case of a reference to a variable it is the fully qualified name. That means the following: If it is a reference to a component of a structure, then the identifier list consists of the main identifier of the complete aggregate, followed by the qualifying identifiers of all substructures containing the referenced component. If the concrete program does not specify the fully qualified name, the translator completes it by inspecting the corresponding declaration.

(2) The pointer qualifier. This component may exist only in the case of a reference to a based variable. It is itself a reference referencing the qualifying pointer.

(3) The subscript list, consisting of expressions and possibly asterisks. In the concrete text of any array reference, the subscripts for the individual array dimensions may be arbitrarily added to any identifiers of the qualified name. Furthermore, subscript lists for array references and argument lists for function references are syntactically not distinguishable. The translator inspects the corresponding declaration and collects all array subscripts, but not function arguments, into the subscript list. Non-array references have the empty list as subscript list.

(4) The argument part. It contains the arguments in the case of a function reference. Since a function may return the value of an entry constant which may be invoked again with another argument list, the argument part is not just a list of arguments, but a list of argument lists. An argument list is a list of expressions. A single argument list or the complete argument part may be the empty list. The latter is the case for all non-function references. It should be noted, that a reference may well contain both a non-empty subscript list and a non-empty argument part (if a component of an array of entry variables is to be invoked as a function).

<u>Example</u>:

Given a declaration

```
DCL 1 S(M) BASED,
      2 A,
        3 E(N,N) ENTRY(FLOAT,FLOAT) RETURNS(ENTRY RETURNS(ENTRY)),
        ...;
```

the reference

```
P->S(1,*).E(N-1)(X+Y,1)()(0,U*V)
```

is translated into the following object:

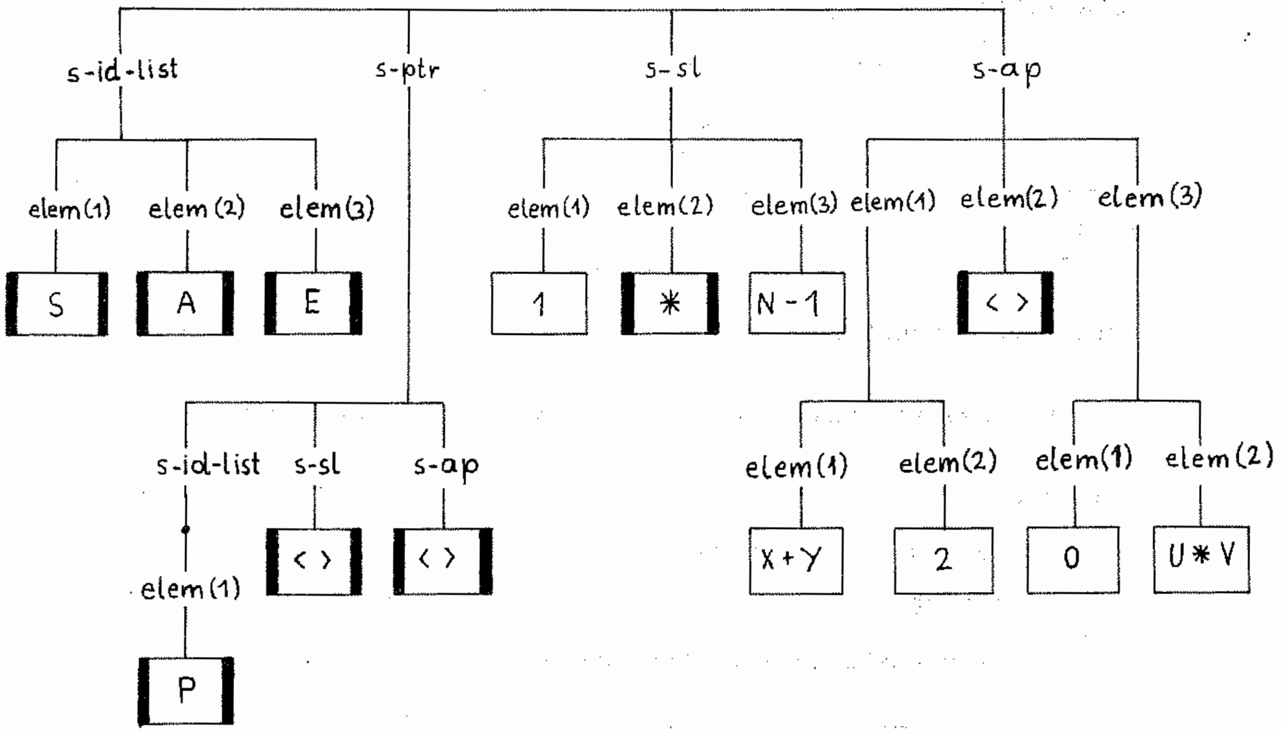


Fig. 2.24 Example of a reference

2.5.2 CONSTANTS

There are two different kinds of occurrences of constants in a concrete program:

(1) in positions where only (signed or unsigned) integer constants may occur (e.g., as precision of an arithmetic data attribute),

(2) as special cases of expressions.

In the first case, the abstract program contains just the value of the constant, which is an elementary object satisfying the predicate is-intg-val.

In the second case, the translator produces an object consisting of the scalar data attributes implied by the form of the constant, and of its value.

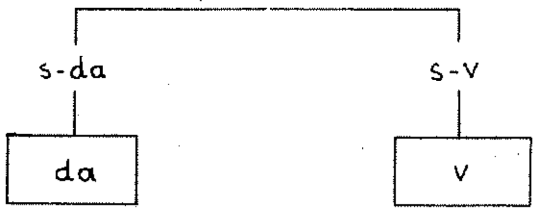


Fig. 2.25 Constant

The data attributes may only be arithmetic or fixed length string.

Example:

The concrete constant

007.30

is translated into the following object:

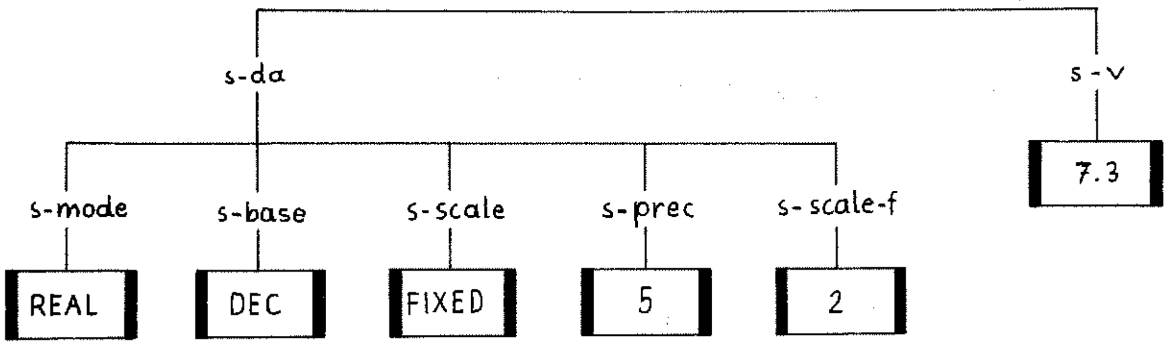


Fig. 2.26 Example of a constant

3. THE STATE OF THE PL/I MACHINE

Corresponding section of /5/:

3. State components and computation of the PL/I machine

The present chapter deals with three aspects under which one can consider the components of the state, namely their use, their scope, and their structure. There are 13 different immediate components of any state of the PL/I machine. One of these, the parallel action part, contains for every task being executed 10 components which have to be considered.

The following is the key to the abbreviations used for the immediate components:

S	the storage
ES	the external storage
UN	the unique name counter
AT	the attribute directory
DN	the denotation directory
FU	the file union directory
TD	the time and date part
ET	the event trace
M	the message part
AN	the attention directory
EV	the attention environment directory
TN	the current task-event name
PA	the parallel action part

There is always one task, the current task, which is currently being executed. Those components of PA which refer to the current task are abbreviated as follows:

TE	the task-event specification
AG	the aggregate directory
FD	the file directory
EN	the enabling state
BA	the block activation name
EI	the epilogue information
CS	the condition status
D	the dump
CI	the control information
C	the control

3.1 THE USE OF THE STATE COMPONENTS

(1) tasking: PA, TN, TE, ET

The parallel action part PA contains for each parallel task or input/output event its local components, i.e., the state components which are used only by the specific task or event. Each task or event has got a unique name, which is used to retrieve its components of PA. The component TN contains the name of the task currently being executed.

The task- and event specification TE contains all information necessary for proper control and, in particular, for the termination of the current task or event.

The event trace ET is a record of all completions of event variables and all starts of executions of wait statements in the current task.

(2) block activations: D, EI, BA

The dump D is a stack which reflects the dynamic nesting of block activations and keeps on each level the state components local to one block activation. The epilogue information EI contains all information necessary to terminate the current block activation correctly. BA contains the unique name given to the block activation.

(3) interpretation of statement lists: CI, C

The control information CI is a stack which reflects the dynamic nesting of groups (statement lists) within the current block activation. The control C can be considered to be a generalized stack, namely a tree which contains the relevant instructions to be executed for the statement currently under interpretation.

(4) meaning of names: DN, AT

The denotation directory DN and the attribute directory AT determine completely the meaning of the declared entities of a program. It is a notable property of these two directories that entries once made are never changed or deleted during subsequent interpretation.

(5) variables: AG, S

The aggregate directory AG and the internal storage S are devoted entirely to the variables of a PL/I program. In particular, AG contains the generations of a variable, which determine the access to S for retrieving the values of the variable.

(6) input - output: ES, FU, FD, M

The external storage ES actually contains the data sets and may therefore be considered as the counterpart to the internal storage S. The two directories FU and FD are entirely devoted to the internal organisation of files.

The message part M is the repository for messages and comments.

(7) unique name generation: UN

The PL/I machine generates unique names during interpretation for identifying uniquely certain pieces of information.

The component UN is just a natural number which determines the next unique name to be used. UN is increased by 1 whenever a unique name is generated, but never decreased.

(8) conditions: CS

The condition status CS contains the information as to which conditions are enabled and which actions are to be performed if a condition occurs.

(9) attentions: AN, EV, EN

The attention directory holds the information relevant for executing access statements and asynchronous interrupts. The enabling state of attentions is kept in EN, the evaluated attention environment in EV.

(10) time and date: TD

This component consists essentially of two integer values specifying the current time and date.

3.2 THE SCOPE OF THE STATE COMPONENTS

This criterion associates each state component with specific sections of the computation. These sections indicate the lifetime of the respective components. Three different scopes are distinguished.

A state component is called program local (of global) if it belongs to the entire interpretation of a program, task local if it is a private state component of a specific task, and block local if it belongs to a specific block activation.

The following lists the state components according to their scope:

(1) program local:

S, ES, UN, AT, DN, FU, TD, ET, M, AN, EV, TN, PA

(2) task local:

TE, AG, FD, EN

(3) block local:

BA, EI, CS, D, CI, C

3.3 DIRECTORIES AND STACKS

Directories and stacks are two important structures of state components.

(1) directories:

A directory is a collection of an arbitrary number of entities each of which consists of a name and some associated information. The name is unique within any directory so that the associated information can be retrieved unambiguously.

The following state components, and the data set directory of <u>ES</u> are directories in the above sense:
<u>AT</u>, <u>DN</u>, <u>FU</u>, <u>AN</u>, <u>EV</u>, <u>PA</u>, <u>AG</u>, <u>FD</u>

(2) stacks:

A stack always reflects some parenthesis-structure. The following three components are stacks:
<u>D</u>, <u>CI</u>, <u>C</u>.

<u>D</u> reflects the dynamic nesting of block activations and <u>CI</u> reflects the dynamic nesting of statements within a block activation.

A stack is a completely ordered sequence of entities. <u>C</u> is a generalized stack in the sense that it represents only a partial ordering.

Among other things <u>C</u> reflects the parenthesis-structure of expressions during their interpretation.

4. STORAGE AND DATA

4.1 REPRESENTATION OF DATA IN THE PL/I MACHINE

Corresponding sections of /5/:

9.1 Values, value representations, operands and operators

This section describes how data is represented in the PL/I machine. Whereas the primary mathematical concept is that of a value, data actually appears in the state components of the machine as value representations and operands.

The introduction of value representations is suggested by some particular features of PL/I which make it advantageous to distinguish between a value and its representation in storage:

(1) A value cannot always be stored and retrieved without loss of accuracy.

(2) By means of based variables a value may be stored with one aggregate attribute and taken out of storage with another (possibly incompatible) aggregate attribute.

(3) By means of record I/O and area assignment storage may be manipulated (independent of data attributes) which can cause an attempt to assign values to or take values from storage with incompatible aggregate attributes.

In both the latter cases undefined situations may arise.

An operand consists of an evaluated aggregate attribute and a value representation, where the attribute constitutes the information necessary for obtaining the corresponding value from the value representation part of the operand. Operands have been introduced because many operations depend not only on the values but also on the attributes of their arguments.

4.1.1 VALUES, VALUE REPRESENTATIONS, OPERANDS

Corresponding sections of /5/:

9.1.1 A class of data attributes

9.1.2 Values

9.1.3 Representing and retrieving scalar values

There are different types of values; they are associated with different types of scalar data attributes which in addition satisfy the predicate is-correct-eda (cf. 9-3(1) of /5/) and they are tabulated in Fig. 4.1:

data attributes	associated types of values
arithmetic	numeric values
character string, character picture, (decimal) numeric picture	character string values
bit string, (binary) numeric picture	bit string values
POINTER, OFFSET	pointer values (cf. 4.2.1)
TASK	integer values (cf. 7.2)
EVENT	event values (cf. 7.5)
ENTRY, FILE, LABEL	unique names (cf. 8.3, 5.5)

Fig. 4.1 Scalar data attributes and their associated types of values

A <u>numerical value</u> is a real or complex number (and it is sufficient to admit only rational numbers, and complex numbers with rational real and imaginary part). A <u>character string value</u> is a list of character values, a <u>bit string value</u> a list of bit values. Examples of <u>character values</u> are the objects A-CHAR,B-CHAR,..., 0-CHAR,1-CHAR,... ; the two <u>bit values</u> are the objects 0-BIT and 1-BIT. Note that 1,1-CHAR,1-BIT (the number 1, the character 1, the bit 1) are different objects. For the remaining types of values, see the sections referred to in Fig. 4.1.

The internal storage of the PL/I machine is used to represent values. Hence, the storage and its constituent parts are called value representations. Interpretation of a reference to a variable eventually leads to an application of the pointer value p to the storage stg (the latter is itself a value representation), which yields a value representation; similarly, interpretation of an assignment to a variable leads eventually to an application of the function el-ass(vr-1,p,vr) which changes the part p(vr) of the value representation vr to the value representation vr-1 (cf. 4.2).

The sense in which a value representation represents a value is explained in the next section; a value v is always represented with a given evaluated aggregate attribute eva , and to retrieve v from its representation, eva is needed again.

Frequently, not only the values, but also the attributes of data are needed. An <u>operand</u> is an object consisting of two components, an evaluated aggregate attrubute eva and a value representation vr (Fig. 4.2). The eva-part of the operand consists of an evaluated aggregate attribute and a density (cf. 2.2.1).

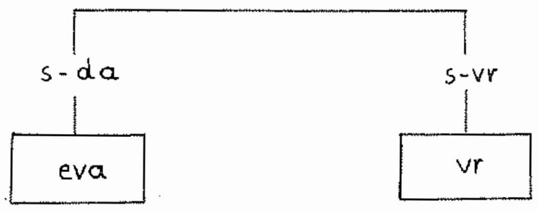


Fig. 4.2 Operand

The result of the evaluation of an expression and the arguments for many operations (infix operators, prefix operators, most of the built-in functions, conversion) are operands.

The eda-part of an operand may be an area attribute. In this case, the vr-part depends on the allocations made in the area. There is, however, no need to introduce a concept of "area value"; therefore, the area case does not appear in Fig. 4.1.

4.1.2 THE TRANSITION BETWEEN A VALUE AND ITS REPRESENTATION

In this and the following sections, whenever the evaluated aggregate attribute under discussion is held constant it will be denoted by EVA and its corresponding eda-part by EDA (to distinguish it from eva and eda respectively). Let EDA be an evaluated scalar data attribute of one of the types listed in Fig. 4.1 and satisfying the predicate is-correct-eda (cf. 9-3(1) of /5/). The transition between a value v and its representation (with the given attribute EDA) is illustrated by Fig. 4.3:

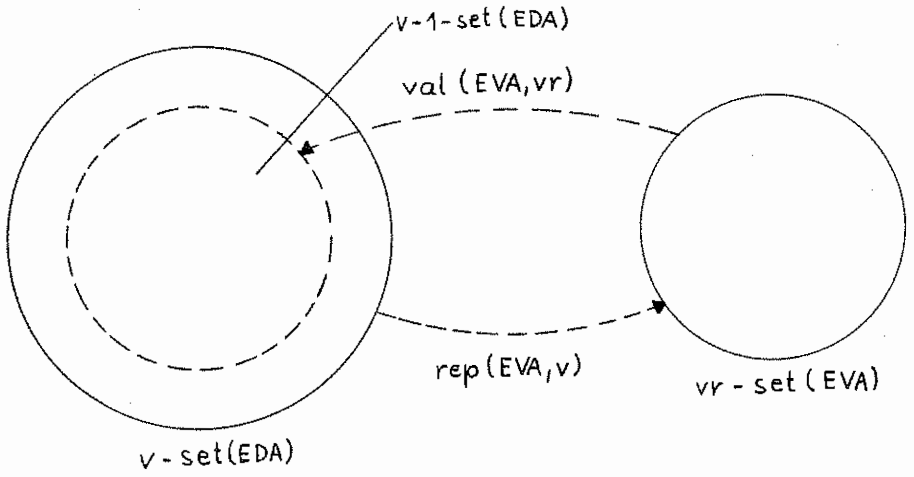


Fig. 4.3 Transition between a value and its representation

The set v-set(EDA) is the set of values which are <u>representable</u> with EDA; these are required to be of the type associated with EDA. The set vr-set(EDA) is the set of value representations that represent values with EVA. The function rep(EVA,v) transforms each element v of v-set(EDA) into an element vr of vr-set(EDA), called the <u>representation</u> of v with EVA; conversely, the function val(EVA,vr) transforms each element vr of vr-set(EVA) into an element v of v-set(EDA), the <u>value</u> of vr with EVA.

Consider the set v-1-set(EDA) of values that are assumed by val(EVA,vr) if vr ranges over vr-set(EVA); this is a subset of the set v-set(EDA). The following is postulated about the functions rep(EVA,v) and val(EVA,vr) (cf. 9-6(18),(19) of /5/):

> For all elements vr of vr-set(EVA) and all elements v of v-1-set(EDA) the two relations
>
> vr=rep(EVA,v) and v=val(EVA,vr)
>
> are equivalent; i.e., rep(EVA,v), considered as function over v-1-set(EDA) only, and val(EVA,vr) are inverse functions.

In view of this postulate, v-1-set(EDA) can be called the set of <u>exactly representable</u> values, i.e., of those values for which transition from a value to its representation, and the representation back to its value, results in the unchanged value. (However, those values not in v-1-set(EDA) are not exactly representable, because the function val(EVA,vr) always leads into v-1-set(EDA)).

<u>Examples</u>:

(1) Let EDA be the attribute CHAR(4). The set v-1-set(EDA) of exactly representable values is the set of all character strings of length 4, whereas the set v-set(EDA) is the set of all character strings. Hence, the string 'ABCD' will be exactly representable, whereas the strings 'ABC' or 'ABCDE' will not; the values of the representations of the latter two strings will be 'ABCb' and 'ABCD', respectively (where b denotes blank).

(2) Let EDA be REAL DEC FIX(4,1). The set v-1-set(EDA) is impementation defined, the set v-set(EDA) is the set of all numeric values that will not raise the SIZE condition. It will be guaranteed that the number 123.4 belongs to v-1-set(EDA), but not, that the number 123.45 belongs to v-1-set(EDA); the value of the representation of the latter may be 123.45, but it may also be 123.4, or 123.5 or something else.

In some cases, a test has to be made as to whether the value is representable; if it is not, the SIZE, STRZ or CONVERSION condition will be raised. For these cases, an instruction <u>test-rep</u>(eva,v) is defined instead of a function rep(eva,v). For representable v, <u>test-rep</u>(eva,v) behaves like a function in that its only effect is to yield the representation of v; it is this function, whether it is called rep(eva,v) or not in /5/, which is meant in Fig. 4.3.

For string EDA, the v-1-set(EDA) is the set of character string values, or bit string values, whose length satisfies the requirements prescribed by EDA. The set v-set(EDA) is the set of all character string values, or bit string values, and when necessary, v is transformed into an element v-1-set by truncation, or by extension with BLANK or 0-BIT respectively. Strings are represented linearly in storage (cf. 9-7(24) of /5/ and 4.2.7.3).

For real arithmetic EDA, let b be the radix of EDA, i.e., b=10 or b=2 depending on whether the base of EDA is decimal or binary; let p be the precision of EDA, and, for fixed-point EDA, let q be the scale factor of EDA.

The set v-set(EDA) of representable real numerical values is the set of all real numerical values v such that

(1) for fixed-point EDA: $|v.b\uparrow q| < b\uparrow q$

(2) for floating-point EDA: $\text{min-flt-EDA}_1 \leq v < \text{max-flt-EDA}_1$

where min-flt-EDA_1 and max-flt-EDA_1 are certain implementation-defined limits (depending only on b) (cf. 9-12(62),(63) of /5/).

The set v-1-set(EDA) of exactly representable values is implementation-dependent. It will, however, contain the subset v-0-set(EDA) (cf. Fig. 4.4.1) defined as follows: v-0-set(EDA) is the set of all values v of v-set(EDA) such that

(1) for fixed-point EDA: $v.b\uparrow q$ is an integer

(2) for floating-point EDA: $v=m.b\uparrow e$,

where m and e are integers and $|m| < b\uparrow p$.

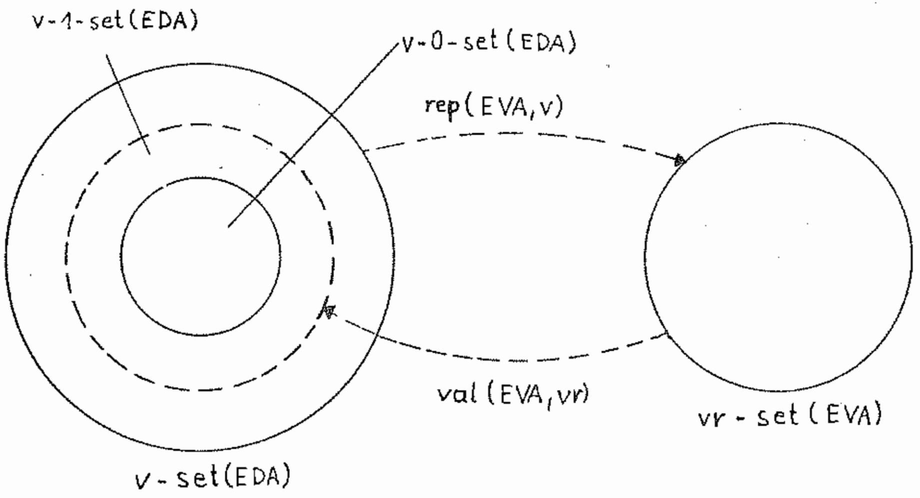


Fig. 4.4a Transition between a real value and its representation

The definition of v-0-set(EDA) expresses that p is "the number of digits" and q is "the number of digits to the right of the decimal point". Since no particular normalization rule is assumed, the limits for floating-point representation (cf. the definition of v-set(EDA)) have been expressed as limits for the entire value, not for the exponent.

Example:

Let EDA be REAL DEC FIX(4,1) (cf. exampl 2 in 4.1.2). The set v-set(EDA) of representable values is the set of real numerical values v such that |v| < 1000. The set v-0-set(EDA) of values for which exact representation is guaranteed is

{0,+0.1,-0.1,+0.2,-0.2,...,+999.8,-999.8,+999.9,-999.9}.

But v-1-set(EDA) may be larger than v-0-set(EDA), i.e., there may be other values (within v-set(EDA)) that are exactly representable as well.

For the other attributes EDA, both v-1-set(EDA) and v-set(EDA) are the set of all values whose type is associated with EDA, i.e., all these values are exactly representable (Fig. 4.4.2):

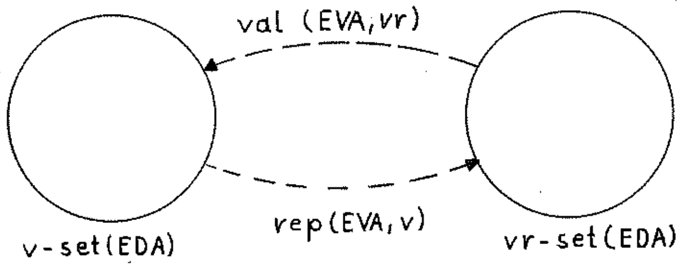


Fig. 4.4b Transition between a non-real value and its representation

4.2 INTERNAL STORAGE AND GENERATIONS OF VARIABLES

Corresponding sections of /5/:

3.4 The storage part S

3.3.4 The aggregate directory AG

8.3.4 Sub-generations of generations

4.2.1 STORAGE AND STORAGE PARTS

The storage part S of the PL/I Machine is a model of actual computer storage. It shows, however, only the essential properties which may be attributed to any actual storage, without exhibiting any properties specific to a particular realization. No explicit construction of the storage part is therefore given. It is rather described by the properties of and the relations between the functions which perform the basic actions on the storage part. This descriptive method while still being precise frees the definition from the burden of unnecessary details.

Storage parts are used to represent values of some kind. They are called, therefore, value representations. Even the entire storage part S is said to be of the type value representation.

There are functions which select parts out of a value representation, which are called pointers. Given a value representation vr and a pointer p we call p(vr) the p-part of the value representation vr. If vr has no p-part, p(vr) is undefined and we say that p is not applicable to vr.

The characteristic property of a value representation is its size. The size of a value representation determines which pointers are applicable to it and, consequently, which parts one may select from it. In turn, the size of a part selected by a pointer is determined uniquely by the pointer (i.e. provided that vr has a p-part, the size of p(vr) is independent of any further properties of vr).

A two-dimensional picture may illustrate the relation between a value representation vr and its parts:

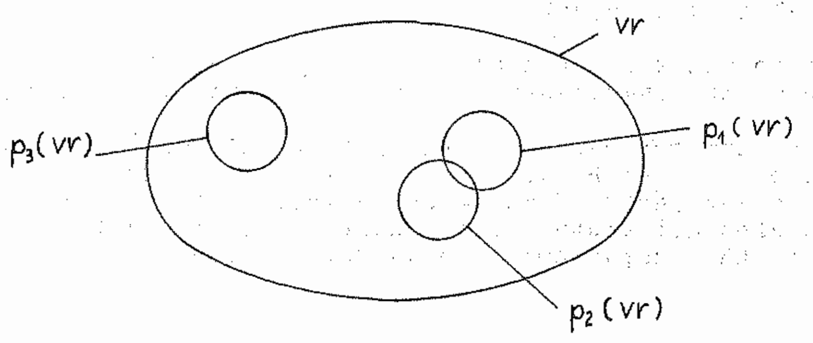


Fig. 4.5 Two-dimensional picture of storage

If two pointers select independent parts, they are called <u>independent</u> pointers. Two parts being independent means that they have no parts in common. Non-independent parts are shown in Fig. 4.5 by overlapping regions, i.e. p_1 is independent of p_3, but not independent of p_2.

<u>Example</u>:

A linear bit storage may serve as concrete model. A value representation consists of a linear arrangement of single bits, indexed from 1 up to a maximum index n. n is the size of the value representation. Each pointer is a function with two integer arguments $f(i_1,i_2)$. It is applicable to a value representation vr of size n if $1 \le i_1 \le i_2$ and $i_2 \le n$. $f(i_1,i_2)(vr)$ denotes the part between and including the i_1th and the i_2th element. Two pointers $f(i_1',i_2')$ and $f(i_1'',i_2'')$ are independent if $1 \le i_1' < i_2'$, $1 \le i_1'' < i_2''$, and either $i_2' < i_1''$ or $i_2'' < i_1'$.

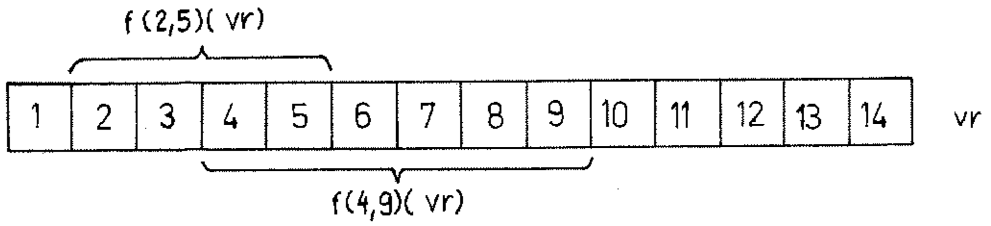


Fig. 4.6 Linear storage model

The p_1-part of the p_2-part of a storage vr is defined by $p_1(p_2(vr))$, or $p_1 \bullet p_2(vr)$. The symbol $\bullet$ is used for functional composition. The term $p_1 \bullet p_2$ represents again a pointer.

4.2.2 ELEMENTARY ASSIGNMENT

The p-part of a value representation vr can be changed by the elementary assignment function el-ass(vr_1,p,vr), provided that the p-part of vr exists, and that the size of vr_1 (the part to be assigned) is equal to the size of the p-part. The function gives a new value representation vr' :

$$vr' = el\text{-}ass(vr_1,p,vr)$$

vr' has the same properties as vr except that vr_1 is now the p-part of vr'. All parts which are independent of the p-part remain untouched.

This has an important consequence, namely that all parts which are not independent of the p-part may be different after the assignment. Since no relationships between parts of value representations are defined, an assignment simply makes all those parts unknown which are not independent of the part to which the assignment is made (with the exception of this part itself).[1]

---

1) There are exceptions to this in string assignment.

Example:

Let the parts of the value representation in Fig 4.5 be:

$p_1(vr) = vr_1$
$p_2(vr) = vr_2$
$p_3(vr) = vr_3$

After execution of the elementary assignment el-ass(vr_1', p_1, vr) the situation is:

$p_1(vr') = vr_1'$
$p_2(vr') =$ unknown
$p_3(vr') = vr_3$

4.2.3 ELEMENTARY ALLOCATION AND FREEING

On allocation of a variable a certain storage part is reserved for holding the values of the variable. The pointer identifying this part is noted in order to prevent further allocations from using it. On freeing, the part is released for further use.

An allocation can be made either in the main storage $\underline{S}$, or (for based variables) in an area (which itself is part of the main storage). The set of pointers identifying those parts of the main storage or of an area which already have been used for allocations, is called the allocation state of the main storage or of the area, respectively. The allocation state is kept in the main storage, or in the area, itself, and can be retrieved by applying the function alloc-state:

allst = alloc-state(vr)

allst is the allocation state of vr (which is the main storage, or an area).

If the p_1-part of $\underline{S}$ or of an area is used for a new allocation, p_1 is added to the respective allocation state. This action is called elementary allocation. Deleting a pointer from the allocation state is called elementary freeing.

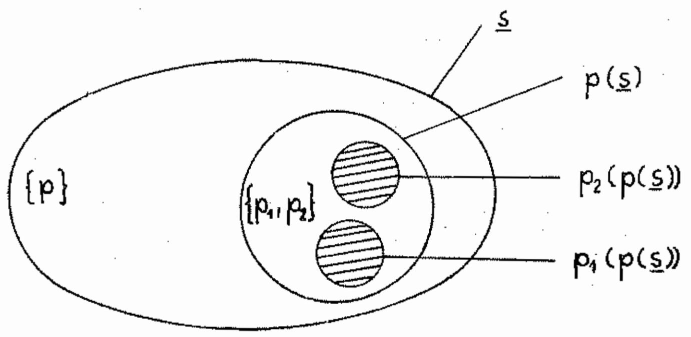


Fig. 4.7 An area is allocated in the p-part of $\underline{S}$. The p_1-part and the p_2-part of the area have been used for allocations. p is noted in the allocation state of $\underline{S}$, p_1 and p_2 are noted in the allocation state of the area.

Elementary allocation and freeing is performed by the elementary allocation and the elementary freeing function, respectively:

el-alloc(p_1,vr) gives a value representation vr' having the same properties as vr, except that its allocation state is amended by p_1,

el-free(p_1,vr) gives a value representation vr' having the same properties as vr, except that p_1 is no longer member of the allocation state.

The storage part which is reserved on allocation of a variable is identified by a pointer given by an implementation-defined function which depends on the evaluated aggregate attribute of the variable and on the properties of the storage (the main storage or an area) in which the allocation is made (for allocations in an area only the allocation state of the area is significant, but not its size). The selected part must fulfil the following requirements:

(1) its size must be such that it matches the size of the value representations which may be associated with the variable;

(2) it must be independent of all the storage parts which are identified in the allocation state of the storage in which the allocation is made;

(3) it must be independent of the part in which the allocation state is kept.

If no storage part having the above properties can be identified, the allocation is not possible. This situation is called <u>storage-overflow</u>. The actions performed on overflow of the main storage are implementation defined. On overflow of an area, the AREA condition is raised (if enabled).

The right size of a storage part is discussed in terms of the storage mapping function in the next section.

4.2.4 STORAGE MAPPING

The properties of a variable which are of significance in connection with storage mapping are expressed by its aggregate attribute.

A variable, according to its aggregate attribute, may be a scalar, an array, or a structure variable. An <u>immediate component</u> of a variable can be identified by an integer value. For array variables, this integer value must be in the range between the lower and the upper bound of the array, for structures between 1 and the number of immediate components of the structure. A component of a variable is again of scalar, array, or structure type. Components of non-scalar components are identified in the same way as immediate components of a variable. A component of a variable therefore is identifiable by a list of integers, which is called reference list (cf. 4.2.6).

An immediate component of a variable is said to be <u>to the left</u> of another immediate component, if the integer identifying it is smaller than the integer identifying the other component. This generalizes in an obvious way to non-immediate components.

The way in which the various components of a variable are associated with storage is determined by the storage mapping function. Let the storage allocated for a non-scalar variable with aggregate attribute eva be p(S). The storage mapping function map(eva,n) gives a pointer p_n such that the nth immediate component of the variable is associated with the storage part p_n(p(S)). The following requirements must be satisfied by the storage mapping function:

(1) if a variable or the component of a variable is an array or a structure, the storage parts associated with the immediate components must be mutually independent.

(2) for each scalar aggregate attribute there is a certain implementation-defined size of those value representations that may be associated with a variable given this attribute (cf. 4.1.2). The size of the part of S associated with a scalar variable, or the scalar part of a variable, must match the size determined by the scalar attribute.

Peculiarities of the mapping function for the handling of strings in storage are discussed in 4.2.7.3.

4.2.5 GENERATIONS OF VARIABLES

The information necessary for accessing storage via a variable is assembled in the generation of the variable. A new generation is formed on allocation of a variable and remains valid until freeing. A generation is not changed between allocation and freeing.

A generation consists of three parts:

(1) the aggregate attribute part. This part consists of the evaluated aggregate attribute of the variable.

(2) the mapping information. This part gives the necessary input to the storage mapping function and consists of an evaluated aggregate attribute.

There are cases where the aggregate attribute contained in the mapping information of a generation differs in array bounds and string lengths from that in the aggregate attribute part of the same generation. This may occur when the storage addressed by the variable owning this generation has not been allocated via this variable (i.e. for data parameters and defined variables).

(3) the pointer part. This part identifies the storage parts associated with the variable. For generations formed on allocation of a variable it consists of a single pointer. Sub-generations of generations and generations of data parameters may have pointer parts which are structured lists of pointers (cf. 4.2.6).

On allocation of a variable with evaluated aggregate attribute eva a new generation is formed with

aggregate attribute part: eva

mapping information: eva

pointer part: single pointer determined from eva and the properties of the storage part in which the allocation is made (cf. 4.2.3).

A generation is called connected if the storage part it associates with a variable can be identified by a single pointer. This is the case if the pointer part consists of a single pointer, and if all the storage identified by that pointer is used by the variable. The latter condition requires that the array bounds and string lengths in the aggregate attribute part are equal to the corresponding array bounds and string lengths of the aggregate attribute in the storage mapping part. It follows that a generation formed on allocation of a variable always is connected.

4.2.6 SUB-GENERATIONS OF GENERATIONS

Given the generation of a variable and a reference to the variable, the sub-generation can be defined which belongs to the part of the variable referred to. The evaluated sub-generation of a variable is used

(1) when an assignment is made to the referenced part (provided that it is scalar)

(2) when the operand associated with the part is to be evaluated (provided that it is scalar)

(3) when it is passed to the parameter of a procedure(cf. 8.3.1).

In the reference to a variable in the program text (cf. 10.2.5) immediate components of structures are identified by identifiers, immediate components of arrays by subscript expressions. On evaluating a reference, identifiers of structure elements are replaced by the indices of the elements (the number of the elements when counted from left to right) and subscript expressions are evaluated and converted to integer values (except when subscripts are specified by asterisks). The result is a list of integer values and asterisks, which is called the reference list.

Example:

Let a variable X be declared in the concrete text as

```
DCL 1 X(7,2) UNALIGNED, 2 Y BIT(3), 2 Z(5) BIT(5);
```

and a reference to X be

```
X (1,*) . Z(5)
```

then the evaluated reference list is <1,*,2,5>.

A generation and a reference list determine a sub-generation in the following way:

(1) A new aggregate attribute part is formed by a sub-aggregate attribute of the aggregate attribute in the aggregate attribute-part of the generation. It is obtained by successively applying the elements of the reference list (from left to right) to determine immediate components of the aggregate attribute. If

the element of the reference list is an integer value i, then the immediate component is

(a) for arrays the aggregate attribute of the immediate array elements

(b) for structures the aggregate attribute of the ith structure element.

If the element of the reference list is an asterisk and the aggregate attribute is an array, then the result is again an array with the same bounds, but with aggregate attributes of the elements as defined by application of the rest of the reference list to the original element aggregate attributes. An asterisk defines a <u>cross-section</u> of the original array.

(2) A new mapping information is formed. The aggregate attribute in the mapping information of the generation is treated like that in the aggregate attribute part (see above).

(3) A new pointer part is formed by successive application of the elements of the reference list:

(a) If the pointer part consists of a single pointer p and the first element of the reference list is an integer value i, then the new pointer part is

$$(map(eva,i))\circ p$$

where eva is the aggregate attribute of the part of the variable corresponding to p. The new pointer part identifies the storage part corresponding to the ith immediate component of this variable component.

(b) If the pointer part is a single pointer p and the first element of the reference list is an asterisk the result is a list of pointers $p_1,p_2,\ldots p_n$. The variable part corresponding to p must be an array in this case, n being the number of immediate elements of the array. The pointers $p_1,\ldots,p_n$ are given by:

$$p_1 = (map(eva,lbd))\circ p$$

$$p_2 = (map(eva,lbd+1))\circ p$$

$$\vdots$$

$$p_n = (map(eva,ubd))\circ p$$

where lbd and ubd are the lower and upper bounds of the array, ubd - lbd = n - 1, and eva is the array attribute. The remaining reference list then must be applied to each individual element of the list $p_1,\ldots,p_n$ in forming the final pointer part. The result will be a non-connected generation.

(c) If the pointer part is a list and the first element of the reference list is an integer value i, the result is the ith element of the list. The part of a variable corresponding to a list of pointers is always an array.

(d) If the pointer part is a list and the first element of the reference list is an asterisk then the remaining reference list is applied to each element of the pointer list (the result being again a list).

The three parts as defined under (1), (2) and (3) form the sub-generation of the generation, determined by the reference list. A sub-generation may be used in the same way as the original generation.

Example:

Consider the reference to the variable X as presented in the preceding example. The step-wise construction of the aggregate attribute part and the pointer part of the sub-generation determined by the reference is illustrated in the following.

Evaluated aggregate attribute eva of X:

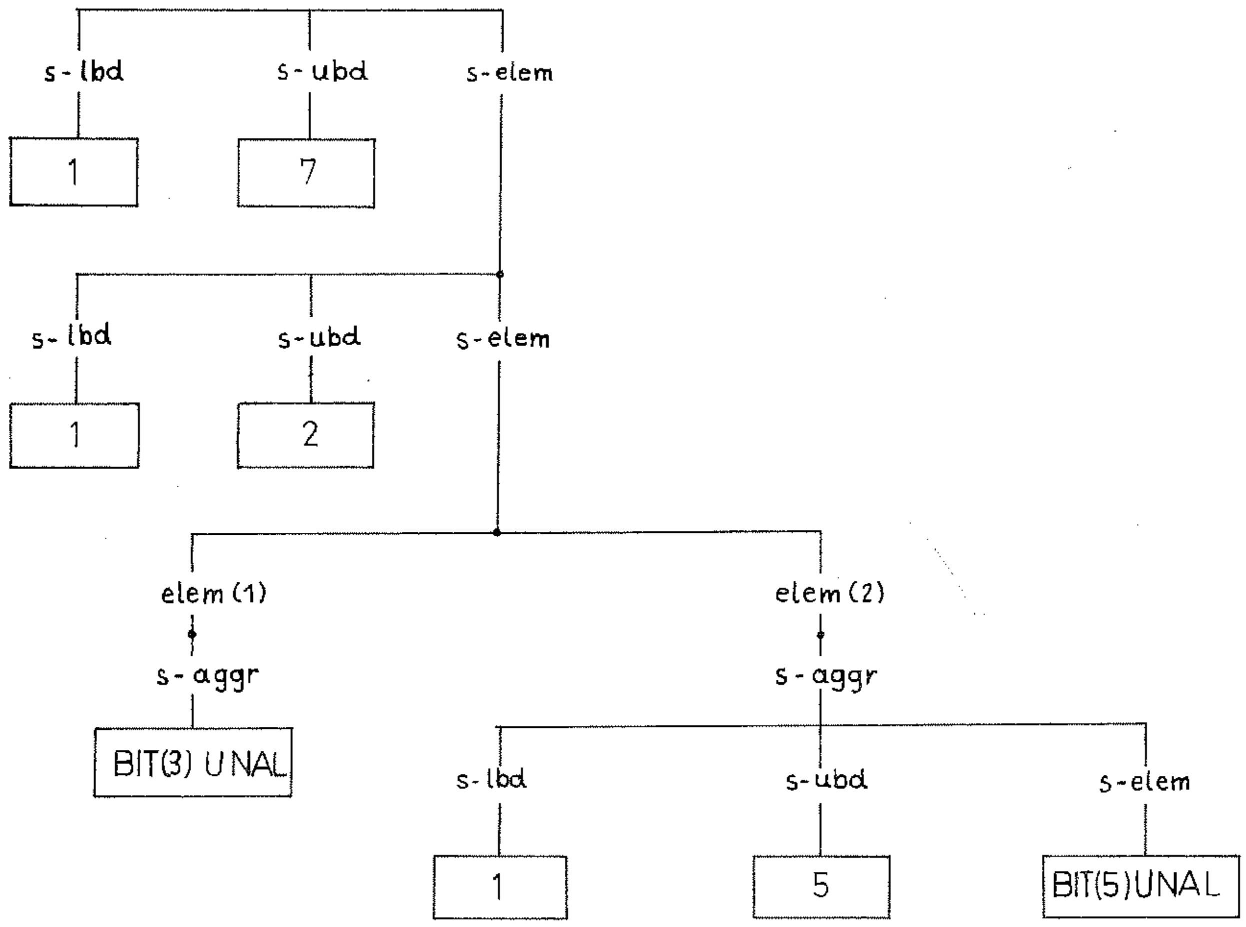


Fig. 4.8a

Let p be the pointer part of the generation associated with X.

(1) The reference list is <1,*,2,5> (see preceding example). The sub-aggregate attribute defined by the first element of the reference list is:

eva_1:

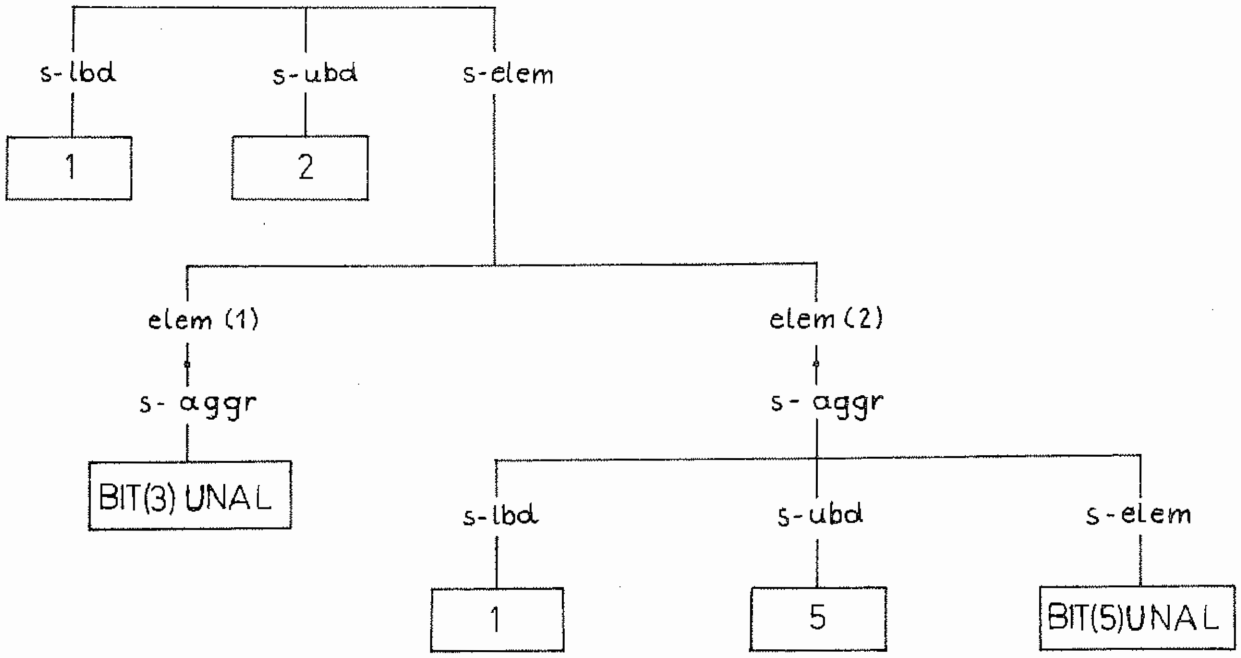


Fig. 4.8b

With p_1 = map(eva,1) we get the modified pointer part: $p_1 \circ p$. $p_1 \circ p(\underline{S})$ is the storage associated with the first element of the array variable X.

(2) The remaining reference list is <*,2,5>. We now have to create a list of pointers, each element corresponding to an element of the array eva_1. With

p_{21} = map(eva_1,1)

p_{22} = map(eva_1,2)

we get the list of pointers: $<p_{21} \circ p_1 \circ p, p_{22} \circ p_1 \circ p>$.

(3) The remaining reference list is <2,5>. The sub-aggregate attribute defined by the first element is:

eva_3:

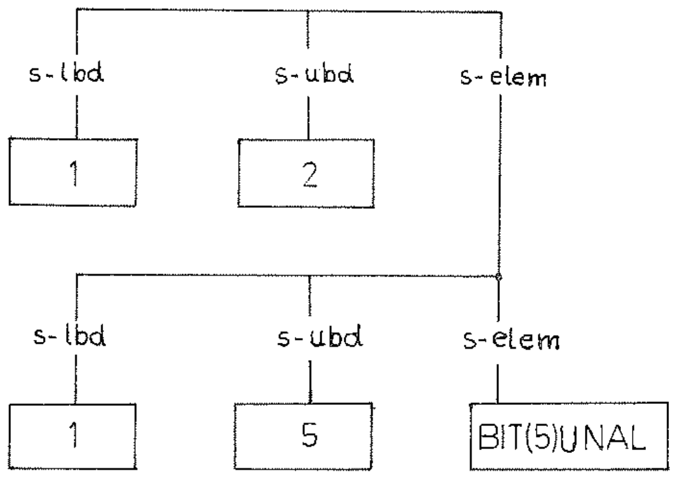


Fig. 4.8c

We now have to modify each element of the above list of pointers. With

$p_3 = map\ (eva_{31}, 2)$

eva_{31}, being the aggregate attribute corresponding to each element of the pointer list (i.e., the elements of the array eva_1):

eva_{31}:

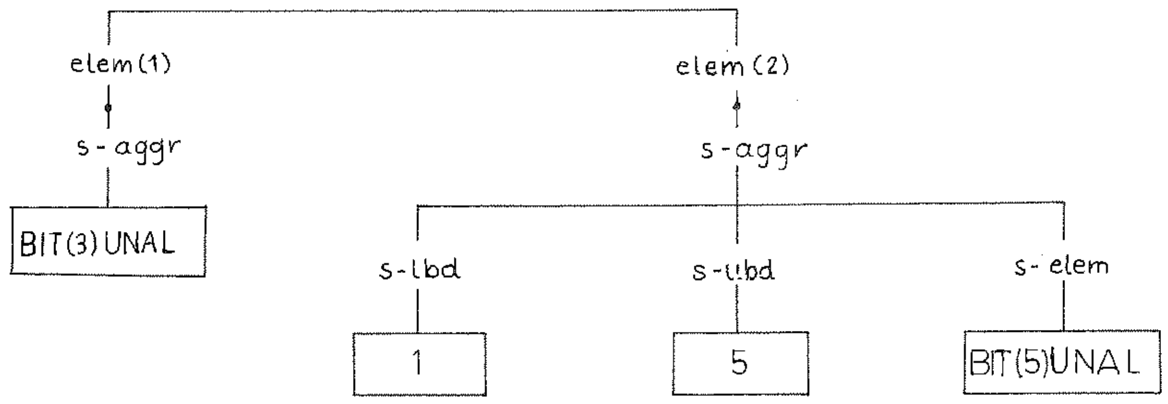


Fig. 4.8d

we get the modified pointer list $<p_3 \circ p_{21} \circ p_1 \circ p, p_3 \circ p_{22} \circ p_1 \circ p>$.

(4) The remaining reference list is <5>. The sub-aggregate attribute defined by it is:

eva_4:

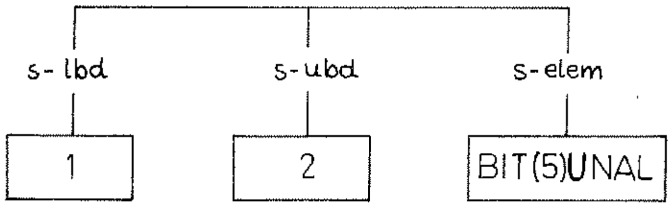


Fig. 4.8e

We have to modify each element of the above list of pointers. With

$p_4 = map(eva_{41}, 5)$

eva_{41} being the aggregate attribute corresponding to each element of the pointer list:

eva_{41}:

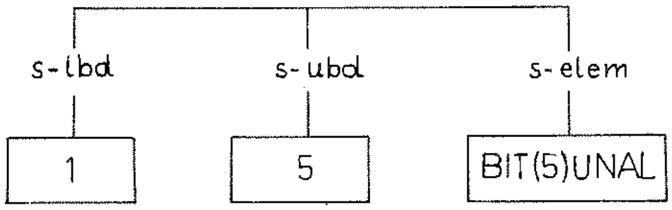


Fig. 4.8f

we get the modified pointer list $\langle p_4 \circ p_3 \circ p_{21} \circ p_1 \circ p, p_4 \circ p_3 \circ p_{22} \circ p_1 \circ p \rangle$.

The resulting sub-generation is composed of the aggregate attribute part eva_4, the mapping information eva_4, and the pointer part consisting of the above pointer list.

Suppose we would use the reference X(1,*).Z(5) as argument to a procedure, where the corresponding parameter P has the attribute eva_4. The non-connected sub-generation corresponding to the reference is then installed as the generation of the parameter P. In case of a reference, say P(2), to the parameter again a sub-generation if formed. This sub-generation consists of

aggregate attribute part: $\boxed{\text{BIT(5) UNAL}}$

mapping information: $\boxed{\text{BIT(5) UNAL}}$

pointer part: $p_4 \circ p_3 \circ p_{22} \circ p_1 \circ p$ (being the second element of the list of pointers in the generation of P).

The operand defined by the reference P(2) consists of the aggregate attribute BIT(5) UNAL and the value representation $p_4 \circ p_3 \circ p_{22} \circ p_1 \circ p(\underline{S})$.

4.2.7 SURVEY OF ATTRIBUTES DEPENDING ON THE STORAGE MODEL

4.2.7.1 Areas

A variable, or part of a variable, declared as an area gets associated on allocation with a storage part, whose size depends on the declared size in an implementation-defined way. A certain part of this storage is always reserved for holding the allocation state of the area. The allocation state is a set of pointers (cf. 4.2.3). Immediately after allocation of an area the allocation state is made the empty set.

Area variables are used to make allocations and freeings via based variables in the storage associated with the area variable (cf. 10.1.1.4).

4.2.7.2 Pointers, offsets

The values of variables declared with the POINTER or OFFSET attribute are pointers as defined in 4.2.1. They can be used to identify storage parts associated with connected generations. Values of pointer variables are used to identify parts of main storage, the values of offset variables are used to identify parts of areas. The use of pointer variables for qualifying references to based variables is described in 10.2.5.3, the use of pointer variables for allocating and freeing via based variables in 10.1.1.

If p(S) is the storage associated with an area, and o is an offset value identifying a part of the area, then $o \circ p$ is the pointer value identifying this storage part in main storage. An area together with an offset relative to this area therefore define a pointer to main storage. Conversely, given the pointer to an area and a pointer to a part of the area, the offset of that part relative to the area can be found. This process is called <u>conversion between pointers and offsets</u>.

It is important to note that an offset value identifying the storage associated with a variable allocated in an area, only depends on the aggregate attribute of the variable and on the allocation state of the area at the time when the allocation was made. The allocation state, in turn, is made up of the offset values identifying those storage parts used by the allocations. Similar allocations made in the same sequence in two different areas therefore define the same allocation state for the areas. An offset identifying the storage part of, say, the last allocation in the one area therefore may be used to identify the storage part of the last allocation in the other area.

The ADDR builtin-function applied to the reference to a variable gives a pointer operand, provided the sub-generation associated with the reference is connected. The value of the operand is the pointer taken from the pointer part of the sub-generation.

4.2.7.3 The ALIGNED and UNALIGNED attributes

Variables may be declared with the attribute ALIGNED or UNALIGNED. These attributes, being part of the aggregate attributes, serve as argument to the storage mapping function (cf. 4.2.4). The intention of unaligned mapping is to optimize with respect to storage space, at the cost of access time. The intention of aligned mapping is to optimize with respect to the access time to the parts of stored aggregates, at the cost of storage space. The exact meaning, however, is implementation defined.

There is a special property of the mapping function for unaligned string aggregates. The location of the various parts in storage is "structure-independent", i.e. the pointer identifying a part depends only on the number of elements (bits or characters) in the part, and on the number of elements (bits or characters) which are to the left of the part in the aggregate. Specifically, the identification of a single bit, or character, is determined by the number of bits or characters which are to the left of it, i.e. by its linear index. This property gives a well-defined relationship between the locations of the elements of two differently structured, unaligned string aggregates. The property is significant for the definition of string overlay defining (cf 10.2.5.2.3).

4.3 EXTERNAL STORAGE AND FILE UNIONS

Corresponding sections of /5/:

3.6 Input and output

11.1 Data set mapping

11.2 Basic access to data

The following abbreviations are used in this section:

mp,MP	mapping parameter
ds,DS	data set
ids,IDS	inner data set or proper inner data set
el	data element or proper data element
vr	value representation
csa	set of file attributes or complete set of file attributes
ea	evaluated environment attribute
f	file name
u	file union name
char	character value
tmt	transmission error flag

File attributes are abbreviated in the text by the first three letters disregarding official keyword abbreviations.
Exceptions: BST is the abbreviation for BITSTREAM, CST for STREAM, and PRT for PRINT.

This section defines the organisation of the external storage which is the repository for data sets and the association of a file with a data set. The association is considered mainly between the file union and external storage. Information concerning the relation between a file value and a file name, and the organization of file directories can be taken from section 5.5 and in more detail from section 12.2.1.

Within the scope of this section it will be of little importance whether a file union name, say u, or the file union itself (i.e., the entry in the file union directory selected by u) is considered. The main difference [1] between the two is that the file union name is valid from the creation of a file union to the end of the computation, and the file

---

1) Another difference is that file union names allow a distinction between identical file unions, which nevertheless might have been created by different openings.

union is valid only until the file is closed, has no impact as long a file is considered after opening and before closing has occurred.

4.3.1 EXTERNAL STORAGE

The external storage consists of two immediate components (Fig. 4.9). One component is the data set directory consisting of data sets and optional transmission error flags. An entry of the data set directory is selected by a data set name. The other component of external storage conveysinformation concerning data set sharing. This component is constant for a particular program but it is implementation-dependent. It reflects the "program-dependency" of the relation between a file and a data set.

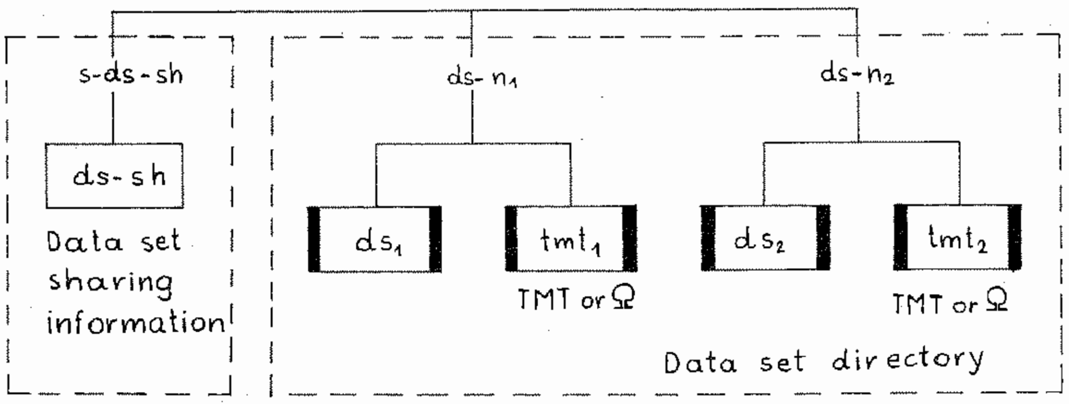


Fig. 4.9 External storage ES

The external storage is initialized by the initial call. Changes of the data sets may occur by data transmission and by indeterministic environmental influences, such as data set switching in case of multiple volume data sets, input on transient (i.e., tele-processing) data sets, transmission errors which set the transmission error flag to TMT, and all kind of operator interference. Environmental influences may cause the insertion of data sets under data set names which previously yielded empty data sets. However, data sets must not be deleted, and a transmission error flag TMT must not be reset to empty.

It is important to note that speaking of a "multiple volume data set", "keyed data set", etc. is only an inexact way of expressing the fact that a data set could have been related (or in fact is related) with a file (or files) treating the data set like a multiple volume data set, or like a data set containing keyed records. That is, data sets do not have a structuring as such: they are elementary objects. However, if a data set is related at some instance with several files, they might treat the data set as if it were structured. At the same instance, the structuring may be different for different files, i.e., data transmitted to the data set by one file may appear differently to another file. A file is said to map a data set into the structured form of an inner data set. Data set mapping depends on the file union and on the data set (cf.[1]4.3. 3).

A file union is always related with exactly one data set which can be conceived as a single volume. The environment attribute and the data_set_title contained in the file union and the data set sharing information of ES are the arguments of an implementation-dependent function which yields the data set name selecting the data set. The case that the data set name refers to no entry of the data set directory is excluded at opening, and no file union will be created in such a case. The data set related with the file union remains the "same" throughout the existence of the file union. Since, in general, there is no privacy of a data set, the only entity which is guaranteed to remain the same in the data set name.

A data set contains data, i.e., entities which explicitly take part in data transmission. In addition, it contains descriptive information about data. For this reason, it would not be instructive to imagine e.g., a record data set as a list of "logical records". A more realistic view supported by the model to be developed would be to conceive of a data set which is composed of the complete contents of the medium on which it is stored (e.g., the dump of a disk pack including all meaningful data separators, and including all hidden buffers) and of the description of the contents (e.g., location of the data set lables, blocking format, physical record length, and keys of records). This concept of a data set is a generalization of the usual notion of a data set.

4.3.2 FILE UNION OF A FILE

The information necessary for accessing the data set associated with a file is assembled in the file union of a file which is an entry of the file union directory FU. The file union is created on opening of a file and remains valid until closing. A file union consists of components which are present in every file union and which remain constant between opening and closing: the "file parameter" and the file name f (Fig. 4.10).

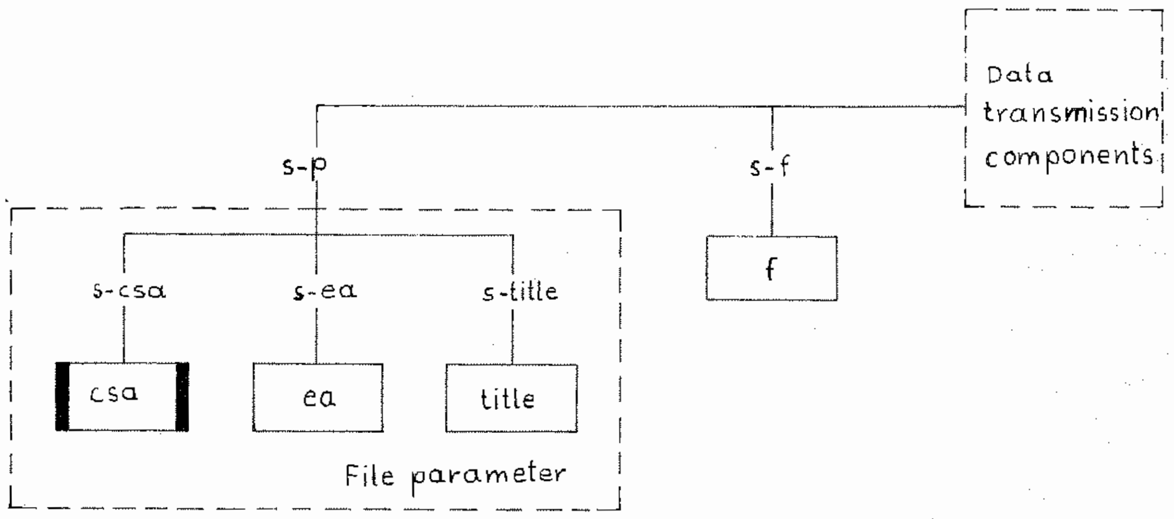


Fig. 4.10 File union

All components of the file parameter are computed on opening. The file parameter components ea and title serve for localizing one data set in <u>ES</u>; the <u>mapping parameter</u> (which is identical with the file parameter except for buffering or exclusivity attributes contained in csa) ascribes a certain structuring to the localized data set.

The component csa of the mapping parameter can be one of the following twenty sets of file attributes:

```
BST with INP or OUT
CST with INP or OUT or OUT,PRT
REC, SEQ with INP or INP, KEY or INP, BAC or INP, KEY, BAC or
              OUT or OUT, KEY or
              UPD or UPD, KEY
REC, TRA with INP or INP, KEY or
              OUT or OUT, KEY
REC, DIR, KEY with INP or OUT or UPD.
```

The other components of the file union are summarized in Fig. 4.11 in tabular form, containing an abbreviation of the name of the component, a description as to whether the component is constant or variable and of what kind of file union it is a part, and references to sections which describe the component in more detail.

Component	Description	Type of file union		
f	copy of file name used for on-condition raising	constant		cf. 12.2.3.2
st	status of the file union with respect to data set switching and data set label processing	variable		cf. 12.2.3.2, 12.2.3.3, 12.5(2)
volno	number of current volume	variable	non-keyed	cf. 12.4
col	current column	variable	stream	cf. 12.2.3.2, 12.6.3.1
count	number of data fields transmitted since start of last statement			cf. 12.6.1, 12.6.2.1, 12.6.3.2
lsz	maximum number of bits or characters in a line	constant	output stream	cf. 12.2.3.2, 12.6.3.1
line	current line	variable	print	cf. 12.2.3.2, 12.6.3
psz	maximum number of lines on a page	constant	print	cf. 12.2.3.2, 12.6.3
buf	buffer pointers with or without key	variable	buffered sequential or transient	cf. 12.3.2, 12.5.3
io-ev	names of attached I/O-events	variable	record non-transient non-buffered	cf. 12.2.3.2, 12.3.2, 12.5.1
tn-key	names of tasks and the keys locked by them	variable	exclusive	cf. 12.5.1, 12.5.2

Fig. 4.11 File name and data transmission components of a file union

The file union contains nearly all information which characterizes a particular "generation" 1 of a file, and the variable components of the file union keep the necessary history. Only task-local information for a file cannot be stored in the file union. This information is part of the file directory and is necessary for opening and closing and for the interpretation of transmission errors on stream files.

4.3.3 DATA SET MAPPING

The necessity of data set <u>mapping</u> originates from the various ways one and the same data set (more exactly: a data set accessible by one and the same data set name) may take part in data trasmission. The concept of mapping is already needed in the case where considerably different

---

1) The unique identification is a file union name

file unions are successively associated with a particular data set. Data set sharing by file unions existing concurrently, irrespective of whether they are shared again over tasks, only influences the logical statements which can be made with respect to the mapping.

Example:

```
  MAIN:PROC ...
       PUT FILE(A) EDIT(X,Y,Z) (A(20));               stream file union
       ...
       CLOSE FILE(A);
       ...
       IF X=Y THEN READ FILE(A) INTO(U);              record input file union
              ELSE OPEN FILE(A) UPDATE KEYED;         record update file union
                   REWRITE FILE(A) KEY(V) FROM(Z);
       ...
       END;
```

The environment attributes for all file unions are the same (they are not specified in concrete text), and also the data set titles are identical ('A'), hence all file unions refer to the same data set name. The output produced with the stream file union may be read in by the record input file union or by the record update keyed file union. As to whether reading or updating is legal, and what would be the effect of reading or updating of the data set, can be decided only at execution of the READ or REWRITE statement. It is anticipated that PL/I guarantees very little about the final state of the data set under consideration in this example.

The file unions a data set is associated with and the "contents" of the data set are not predictable until the data set is effectively accessed somewhere during program execution. This situation can be roughly compared with the assumption that somewhere during execution of a program it might be possible to switch from the 60-character representation of a concrete program into the 48-character representation, or to treat blanks like semicolons. With this assumption it would not be possible anymore to translate a program into abstract text, and the concrete text itself would have to be interpreted.

The situation may also be compared to a value representation being totally or partially accessed by several generations. Only for certain generations can the value representation be mapped into a reasonable value.

If a data set ds can be accessed, i.e., if the mapping exists, the mapped data set is called an inner data set ids. The mapping depends solely on the mapping parameter mp of the file union and the data set:

ids = decipher(mp,ds)

The structuring of inner data sets, and the notion of proper inner data sets is described in the sequel. Since data transmission consists of a change of ds, this change has to be reflected in ES from where ds is taken. The changes are always made explicitly in the ids, and the changed data set which will replace the data set in ES. The mapping from ids back to ds, and the most general properties of both mapping are described in the sequel. Those properties of the mapping specifically related with basic data transmission are discussed in section 4.3.4.

4.3.3.1 Inner data sets

Given a mapping parameter MP, there will be a subset of data sets which can be mapped into inner data sets. This subset is characterized by all data sets ds for which the predicate is-decipherable(MP,ds) is true. If this subset is non-empty,[1] there is at least one data set, say DS, which yields IDS, and this inner data set is in addition a proper inner data set (see below). Not every proper inner data set can be mapped back into a data set. However, IDS can be mapped back and yields DS again. Hence, under the premise is-decipherable(mp,ds), for every mapping parameter mp and data set ds

decipher(mp,ds) = ids and cipher(mp,ids) = ds

is guaranteed.

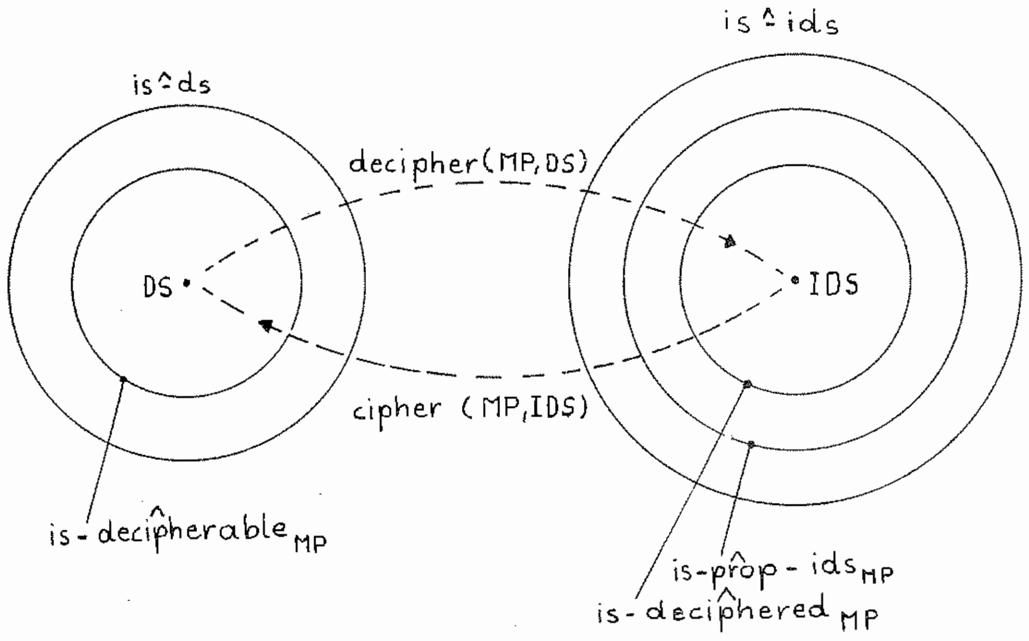


Fig. 4.12 Domain and range of the mapping functions decipher/cipher, given a particular mapping parameter MP

In Fig. 4.12 the sets of data sets, inner data sets and proper inner data sets are symbolized by $\hat{is}$-ds, $\hat{is}$-ids, and is-$\hat{p}$rop-ids$_{MP}$.[2] The domain of the function decipher$_{MP}$ is is-deci$\hat{p}$herable$_{MP}$, its range is is-deci$\hat{p}$hered$_{MP}$. The function cipher is the inverse of decipher with respect to the second argument.

Data sets are elementary objects, inner data sets are composite objects (Fig. 4.13 and 4.14).

---

1) This situation will referred to in the sequel by the term "there exists a mapping".
2) The subscript denotes a dependency on the mapping parameter MP which should be considered fixed for the moment. The set of all objects for which a predicate is true is symbolized by the sign "$\hat{}$" above the predicate name.

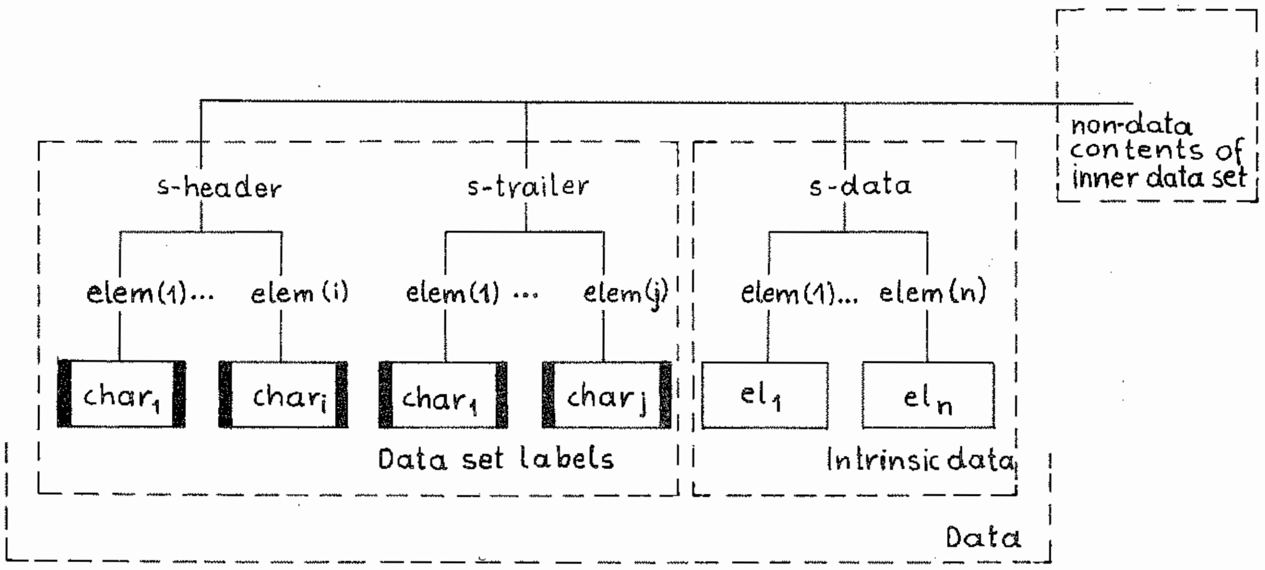


Fig. 4.13 Data contents of inner data set

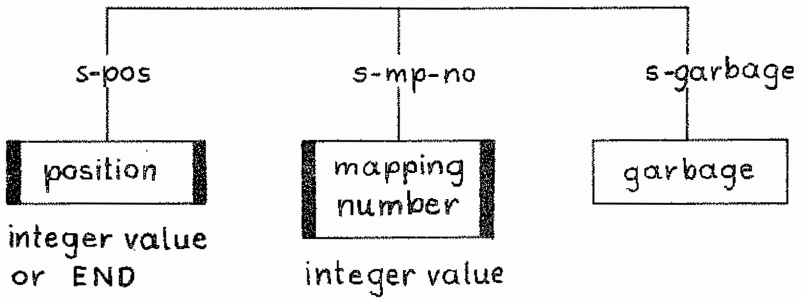


Fig. 4.14 Non data contents of inner data set

All components summarized as data in Fig. 4.13, and the position and mapping number components of Fig. 4.14 are inspected or changed by data transmission. The garbage component comprises all the information which is part of the contents of the data set but which is hidden to the mapping parameter MP under consideration. The garbage may contain for example

(1) the positions and possibly the buffers of the file unions sharing the data set for the moment,

(2) descriptive information about the data set,

(3) hidden information which is not accessible by the file union under consideration but may be transparent to another file union, etc.

File unions with the <u>same</u> mapping parameter sharing a data set yield the same inner data set. The <u>mapping number</u> of the inner data set is the number of the sharing file unions with the same mapping parameter, so to speak the number of equivalent data trasmission paths.

<u>Example</u>:

Fig. 4.15 shows a part of <u>ES</u> and <u>FU</u>, namely two file unions with MP_1 and one with MP_2 as the mapping parameters. If it is assumed that there are no other file unions in <u>FU</u> having MP_1 or MP_2 as mapping parameter, the mapping number of IDS_1 is two and of IDS_2 is one.

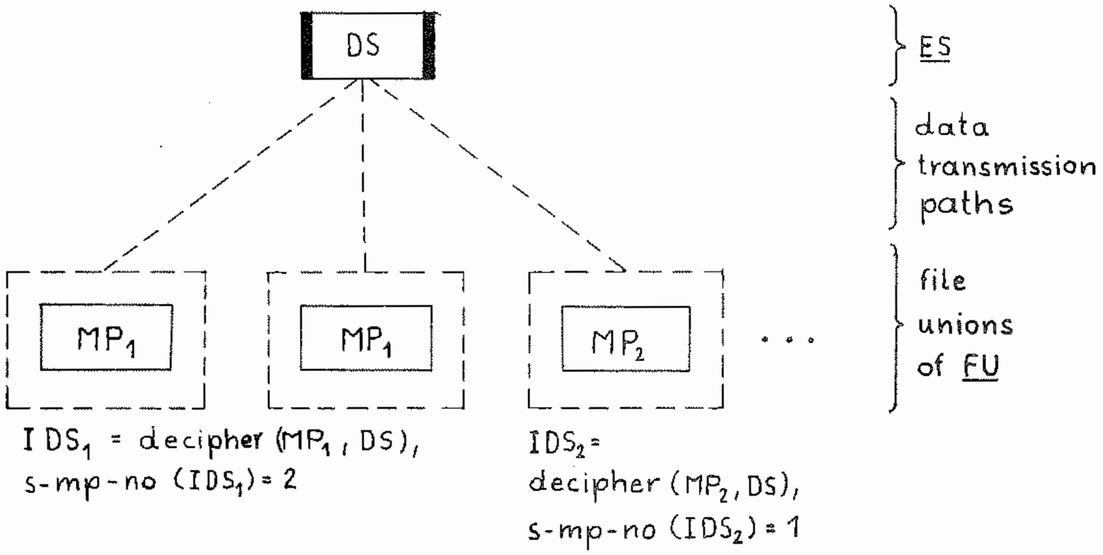


Fig. 4.15 Example for the role of the mapping number

Mapping numbers are adjusted exclusively at creation or deletion of a file union. Notice that the information on the mapping number is containedin the data set and not in the file union (cf. 4.3.3.2).

The <u>position</u> component of the data set denotes the position of the last element of intrinsic data which has been transmitted. The position is END if the end of the data set has been reached by some previous transmission.

The <u>data set labels</u> are lists of character values, the no-label case is modelled by empty lists. The header label is always processed during the data set opening phase (cf. 4.3.3.3).

<u>Intrinsic data</u> consist of stream or record elements (el_1 to el_n in Fig. 4.13). Bit or character values and the elementary objects line-delimiter (LDEL), page-delimiter (PDEL), carriage-return (CRET), and tabulator (TABL) are <u>stream elements</u>. A value representation, a value representation and a key being a list of character values, and the elementary object specifying a deleted record (DELETED) are <u>record elements</u>.

The value representation component vr of a record element usually is an exact copy of the storage being transmitted. The record's vr may serve as the source of subsequent "as-is" assignment to an aggregate,[1] or it may be used to determine a pointer value to accomodate vr if suitable buffer or area storage has to be allocated for vr.

Example:

```
MAIN:PROC ...
     DCL A KEYED,
         1 X, 2 X1 FLOAT(16), 2 X2 CHAR(3), 2 X3 FLOAT(16),
         Y(2,5) CHAR(2);
     REWRITE FILE(A) FROM(X); --> vr,k
     X = ... ;
         READ FILE(A) KEY(k) SET(P);  <-- vr
         READ FILE(A) KEY(k) INTO(Y); <-- vr
         READ FILE(A) KEY(k) INTO(X); <-- vr
     END;
```

The REWRITE statement is assumed to open A as a sequential update keyed buffered file. Transmission of X means updating of the record element b y the storage associated with X, say vr. This record element is assumed to have key k. Any of the following three READ statements is assumed to fetch vr. If there is enough free storage, storage allocation and initialization with vr will be guaranteed for the first READ. Further reference with pointer P will depend on the aggregate attribute associated through the reference. The second READ is only guaranteed if the sizes of vr and of the aggregate referenced by Y are the same. The meaning of further references to Y or sub-aggregates of Y is implementation-dependent (cf. the function map in 4.2.4). The third READ is guaranteed to reestablish X with the meaning it had at the moment when the REWRITE was executed.

A proper inner data set is an inner data set with the following additional properties (the description is based upon Figs. 4.13 and 4.14):

(1) if the position is an integer, it is positive and does not exceed n,

(2) the mapping number is positive,

(3) all data elements $el_1,\ldots,el_n$ are proper data elements or they are DELETED (optional only in the case the attribute REC is contained in the mapping parameter),

(4) all data elements $el_1,\ldots,el_n$ must have a proper key in case the attribute KEY is contained in the mapping parameter.

The decision as to whether a key is proper must be left to the implementation. The correspondence between attributes and proper data elements is given in Fig. 4.16. Notice that the data elements are arranged in a list also in case the mapping parameter contains the attribute DIR.

1) el-ass(vr,target-p,target-vr) cf. 4.2.2

Mapping parameter containing the attributes:	Proper data element is a:
BST with INP or OUT	bit value or LDEL
CST with INP or OUT	character value or LDEL
CST with OUT, PRT	character value, LDEL, PDEL, CRET, or TABL
REC with KEY, etc.	value representation and key
REC without KEY, etc.	value representation

Fig. 4.16 Mapping parameters and proper data elements

4.3.3.2 Data set activity

A data set is said to be <u>active</u> with respect to a certain mapping parameter mp if the corresponding mapping number #mp is greater than zero, inactive otherwise.1 Fig. 4.15 shows a data set DS being active with respect to MP_1, MP_2, etc. The following statements on #mp can be made:

(1) It is guaranteed that an inactive data set can be opened.

(2) Opening increases #mp by one,2 closing diminishes #mp by one.

(3) A change of #mp does not affect data set mapping by active mapping parameters different from mp, except for the garbage component. However, opening with mp may cause that a mapping does no more exist for previously inactive mapping parameters different from mp. Closing with mp may cause the data set to become a candidate for potential mapping and opening by some mapping parameters different from mp. Hence, for any such mapping parameter the mapping number will be zero.

(4) For mapping parameters different from mp, any kind of data transmission including position changes might affect data set mapping in the data, position, and garbage components but not in the mapping number components.

<u>Example</u>:

At some stage of the computation it is attempted to relate a file with a data set DS through opening. Opening is unsuccessful and raises the UNDF on-condition. In the sequel, some files sharing DS are closed. If the first opening is retried subsequently, it might be successful.

---

1) Notice that #mp is a component of an inner data set, hence the existence of a mapping is presupposed in this section.
2) If for an active data set the increased #mp would not preserve the existence of the mapping then opening would not be successful.

Files which do not share a data set have different mapping parameters. Such files have no relation except that they might have the same file name.

4.3.3.3 Forwards and backwards transmission

For the notion of data set activity it was necessary to anticipate parts of the criterion for opening with a particular mp: The existence of the mapping, and the existence of the mapping even for the increased mapping number.

In connection with data transmission obviously only those mapping parameters are of interest which satisfy this criterion.

In particular, for any mp containing the attribute BAC which satisfies the criterion, and for mp_1 differing from mp in the missing attribute BAC only, it is guaranteed [1] that the proper inner data sets yielded by mp and mp_1 have mutually exchanged header and trailer labels, and intrinsic data being arranged in inverted sequence. The position components are the same.

Hence, instead of processing a data set backwards, i.e., decreasing the current position and taking the header label instead of the trailer label and vice versa, the adaption of the above rule of inverted mapping allows getting rid of all these exceptions at once.

4.3.3.4 Related mapping

In general, no conclusion about similarity or dissimilarity of inner data sets can be made if they are yielded by the same data set but different mapping parameters. However, if the mapping parameters are identical except for the set of attributes, and if the attributes neither contain the attributes BAC nor PRT, the inner data sets have the same data (labels and intrinsic data) if all mapping parameters belong to the same mapping category.[2] The mapping category is bit stream, character stream, keyed, and non-keyed if the attributes BST, CST, REC and KEY, and REC but not KEY are contained in the mapping parameter, respectively.

More special relations of mapping parameters containing the attribute UPD with mapping parameters specifying INP or OUT will be detailed in sections 4.3.4.1 and 4.3.4.3.

4.3.4 BASIC DATA TRANSMISSION

Basic data transmission is the main application of data set mapping. Basic data transmission is always performed by "basic groups", i.e., as one elementary step of the computation. The basic groups usually make an explicit change in <u>ES</u>. Depending on the type of basic data transmission

---

1) mp_1 is a "forwards" mapping parameter. The exact formulation of the guarantee is complicated by the fact that from mp satisfying the opening criterion it must not be derivable that mp_1 will satisfy the opening criterion, too. This would be a general statement on the sharing of backwards and forwards mapping parameters which certainly could not be supported by an implementation on tape-like medium.
2) An exception is the mapping category keyed if some mapping parameters contain the attribute DIR and some do not. In this case only identity of proper data elements of intrinsic data is guaranteed but the sequencing is not preserved.

other state components may also be changed. For example, buffers are allocated, freed and assigned to, assignment to the target aggregate of data transmission is performed, etc.

Since this section is devoted solely to the static properties of data set mapping, only the names of the basic data transmitting actions are given, together with references where they are discussed in more detail in this document (Fig. 4.17).

Name of action:	Refer to section	Basic data transmitting function involved:

stream-transmission	12.6.3.1	read(mp,ds,Ω)
		write(mp,ds,el)

into-set-transmission	12.5.3.4, 12.5.3.5	read(mp,ds,key)
set-transmission	12.5.3.4	
ignore-transmission	12.5.3.5	ignore(mp,ds,n)
delete-transmission	12.5.3.5	delete(mp,ds,key)
rewrite-transmission	12.5.3.3	rewrite(mp,ds,el)
write-transmission	12.5.3.1	write(mp,ds,el)
buffer-transmission	12.5.3.1	

Note: el is a proper data element with respect to mp,
key is an optional key, n is an optional integer.

Fig. 4.17 Basic data transmitting actions and functions

The actions are based on the five basic data transmitting functions read, ignore, delete, rewrite and write. All these functions have in common, that they map a data set ds in dependence of additional information (mp, el, key, n) onto a data set ds_1 and some information about the success of this transition inf_1. The resulting data set ds_1 is not necessarily different from ds (Fig. 4.18).

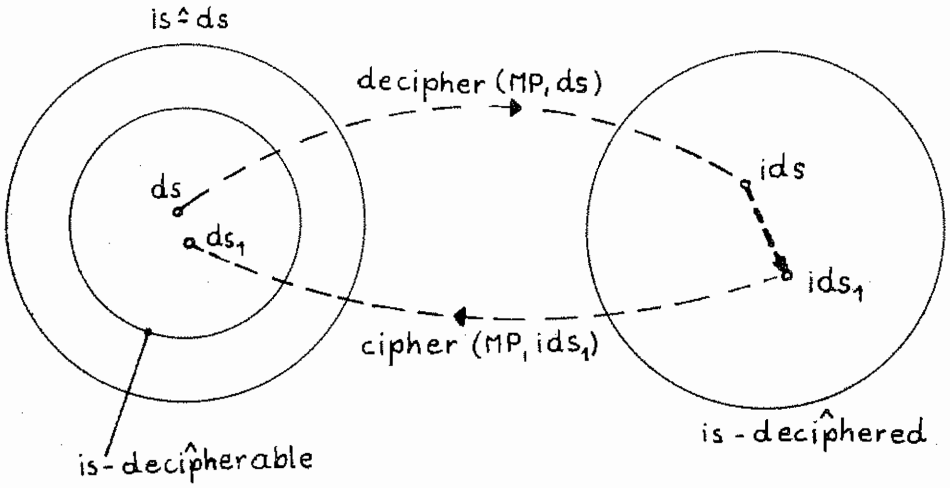


Fig. 4.18 Transition from ds to ds_1 caused by a basic data transmitting function

The resulting information inf_1 is empty or it indicates certain <u>unusual situations</u>, or the proper data element yielded by the function read.

There are three types of unusual situations:

(1) something is wrong with the key, i.e., the key is not proper, or there is no matching key to be found in the case of a read, delete, or rewrite, or there is a matching key found in a write,

(2) something is wrong with the size of the value representation transmitted (rewrite, write),

(3) the end of the data set has been reached (read,ignore,write).

Situations (1) and (2) will give rise to subsequent on-condition calls, situation (3) will cause waiting for input (in the case that mp contains the attribute TRA and INP) or data set volume switching (in all other cases). Hence, unusual situations are detected at the point where basic data transmission is performed. However, the interpretation of unusual situations is not part of basic data transmission.

All basic data transmitting functions become undefined (erroneous) if at least one of the following cases applies:

(a) ds cannot be mapped with mp,

(b) ds is inactive with respect to mp, i.e., the mapping number is zero,

(c) ds is in the position "END" with respect to mp.

Any of the above cases can be true because ds is erroneously shared by mapping parameters other than mp. Since environmental influences may arbitrarily change any data set, such a change may also give rise to an erroneous situation.

If the above kind of sharing and environmental influences are excluded, cases (a) and (b) will not apply since they would prevent opening to occur, and because it is an important property of all basic data transmitting functions that they yield a data set ds_1 which can be mapped again and which is active (also in the unusual situations, cf. Fig. 4.18). Since the basic data transmitting action using the function will replace ds (i.e., the data set as taken from <u>ES</u> before data transmission occurred) by ds_1, the properties of defined mapping and activity are conserved in <u>ES</u>.

Case (c) may apply also due to a sharing of ds with the same mp if basic data transmission over another file union has reached unusual situation (3) previously. This is demonstrated by the following example.

<u>Example</u>:

```
MAIN:PROC ...
     DCL (A,B) RECORD, I INIT(1);
          ON ENDF(A) IF I=1 THEN GOTO E;
          ON ENDF(B) GOTO END;
     OPEN FILE(A), FILE(B) TITLE ('A');
     READ FILE(B) INTO(X);
RDA: READ FILE(A) INTO(Y);
     ...
     GOTO RDA;
E:   I=0;
ENDF:READ FILE(A) INTO(X);
ERR: READ FILE(B) INTO(Y);
END: END;
```

Opening is assumed to produce two file unions with identical mp. If label E is reached then this has been caused by an end-of-file situation on the file union accessible to A. The end-of-file status being registered in the file union is a persisting form of situation (3). The READ statement labelled ENDF is issued in end-of-file status, hence the on-unit is called again without looking at the data set. The READ statement labelled ERR is erroneous since the file union accessible to B has not been set into the end-of-file status, hence the data set is mapped and unusual situation (3) applies.

<u>4.3.4.1 Positioning, reading, and deleting</u>

The basic data transmitting functions read and ignore yield a data set which may differ from the source data set in the positioning only.[1] It is guaranteed that positioning will always be within the range described in section 4.3.3.1 for proper inner data sets, and that the existence of the mapping is preserved. This is true even in those cases where the positioning itself is implementation-dependent because of an unusual situation in connection with positioning by a proper key (cf. 4.3.4, (1)).

---

1) Both functions are not concerned with size violations. Any mismatch between the sizes of the value representation component of the proper data element read and any target storage specified will be interpreted by the action into-set-transmission (cf. Fig. 4.17).

The functions read and ignore, if used for INP file unions, have no peculiarities as compared with their usage in connection with UPD file unions. Hence it may be postulated that mapping parameters differing only in the attributes INP and UPD yield proper inner data sets with identical data provided that the mappings exist and that a potential success of opening with the UPD mapping parameter implies the same success for the INP mapping parameter.

The basic data transmitting function delete changes positioning in a way similar to read and ignore. It is guaranteed that the replacement of the proper data element by the artificial data element DELETED will preserve the existence of the mapping.[1]

4.3.4.2 Rewriting

The proper data element el which has to be transmitted to the data set ds by the function rewrite(mp,ds,el) might violate the implementaion-dependent requirements for the size of the value representation component of el. If it does not violate the requirements then the implementation-dependent predicate is-size-violation(mp,ds,el) is false and el can be rewritten as it is.[2] Otherwise the predicate is-size-violation(mp,ds,el) is true and instead of el another proper data element, say el_1, would be rewritten. The elements el and el_1 are guaranteed to have the same key, and el_1 does not again violate the size reqirements.

Irrespective of whether el or el_1 is rewritten, the existence of the mapping is preserved for the updated data set. p
kind of positioning occurring in unusual situations is described in section 4.3.4.1.

4.3.4.3 Writing

The proper data element el which has to be written in the data set ds by the function write(mp,ds,el) might violate the data set extent and size requirements of the implementation. If both requirements are met then el is written as it is. In this case the implementation-dependent predicates is-end(mp,ds,el) and is-size-violation(mp,ds,el) are both false. However, if the predicate is-end(mp,ds,el) is true no data element will be written but the position will be set to END. This will be done regardless of any size violation.

The last case, is-size-violation(mp,ds,el) being true and is-end(mp,ds,el) being false, is handled in the same way as a size violation at rewriting (cf. 4.3.4.2).

If the attributes BST or CST are contained in mp then the notion of size violation becomes meaningless. In the above description a value of true should be substituted for the predicate is-size-violation though the predicate is not really used in the definition of stream transmission.

---

1) The replacement does not necessarily mean that the deleted proper data element (or parts of it) will not be accessible anymore to mapping parameters other than the mp under discussion.
2) Proper data elements being contained in ds obviously do not violate the size requirements.

The basic data transmitting function write if used for DIR, OUT file unions has no peculiarity as compared with the usage of the function in connection with DIR, UPD file unions.1 Hence it may be postulated that mapping parameters containg the attribute DIR and differing only in the attributes OUT and UPD yield proper inner data sets with identical data provided that the mappings exist and that a potential success of opening with the UPD mapping parameter implies the same success for the OUT mapping parameter.

The kind of positioning occurring in unusual situations is described in section 4.3.4.1.

4.3.4.4 Transmission errors

It has been outlined in section 4.3.1 that the transmission error flag appended to data sets is ES provide the information concerning intervening transmission errors. Setting of the flag is not under the control of the interpreter, and is independent of any environmental change of the data set.

Every basic data transmitting action (cf. Fig. 4.17) inspects the flag and deletes it.

---

1) The function write is not used with the attributes SEQ, UPD.

5. IDENTIFIERS AND THEIR SIGNIFICANCE

This chapter considers a specific aspect of PL/I, namely the kind of names which may occur in a program and their meaning during the execution of that program. The question may be formulated more precisely in terms of the PL/I machine as follows: at some point of time, i.e., for a given state of the PL/I machine, what information is associated with the known identifiers (or identifier lists in the case of qualified references). A diagram of the information which is in general dynamically associated with a name will be given for each specific kind of PL/I name. This will be followed by a discussion as to when the individual components of the diagram are created, changed or deleted. It will also be discussed under which circumstances a name may have parts of the information associated with it in common with another name (sharing patterns). Only single tasks will be considered in this chapter, except for the last section which will give some notes on the consequences of tasking for the subject of this chapter.

The following kind of diagram will be useful in the discussions of this chapter. Whenever it is necessary to say that with a given piece of information A one may retrieve the information B from a directory $\underline{D}$ of the state of the PL/I machine then this is indicated by the diagram:

$$A \xrightarrow{\underline{D}} B$$

In other words A is associated with B in $\underline{D}$.realized in the state of the PL/I machine will, however, be suppressed. The above picture therefore only indicates that it is possible to retrieve B given A from $\underline{D}$ in some way which is not specified further.

Occasionally, it will be necessary to represent composite objects, of some specific kind, in a diagram. This will be done by enumerating variables enclosed in parantheses, where the variables stand for the immediate components of the object and the names of the variables indicate the kind of component. The specific selectors that lead to the components will thus be suppressed.

5.1 DECLARATION AND USE OF IDENTIFIERS

The relation between the use of an identifier and its corresponding declaration in a given program is static, i.e., can be determined without interpretation of the program. The declaration which corresponds to a given use of an identifier is always found in the declaration part of a block containing the use.[1] The innermost block is to be taken in case there is more than one such block.

The initial step of the interpretation of a block is to make a copy of the block. For each declaration of the declaration part a new unique name is created and inserted as an additional component throughout the

---

1) More precisely one should talk about identifier lists corresponding to qualified names rather than single identifiers.

block in any place of use of the identifier. Then a new entity is created for each individual declaration and made available under the corresponding unique name. The term entity means, in this context, a collection of state components which are linked together in some way. In particular, entries in the denotation directory and attribute directory are made for each newly created unique name.

As a consequence, the name of an entity is uniquely associated with a specific interpretation of a specific declaration, i.e., associated with the declaration and a specific block activation.

Throughout the block activation any use of an identifier will be interpreted as references to this entity. block activation be interpreted as references to this entity. It is essential to note that any copy of a part of the block carries the meaning of the identifiers used but not locally declared within that part of the block. Copies of parts of the block which are kept in the state for later interpretation will therefore retain the meaning of their non-local identifiers irrespective of the place where they are executed.

Differnet entities may share components and the present chapter will be partially devoted to the study of these sharing patterns and thereby some properties of PL/I will be formulated.

Fig. 5.1 shows as an example the structure of the simple reference S.A.B, after insertion of the unique name, n say.

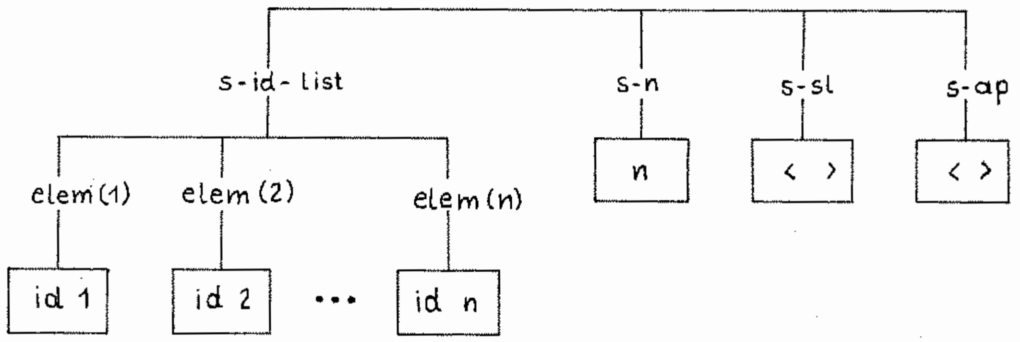


Fig. 5.1 Example for a reference after insertion of the corresponding unique name n

5.2 DENOTATION AND ATTRIBUTES

All entities created by the interpretation of a declaration have a denotation and an attribute part to be found via the unique name in the denotation and attribute directories respectively. The following picture of an entity (Fig. 5.2) is therefore valid independently of the kind of its declaration.

$$\begin{tikzcd} & \mathrm{den} \\ n \arrow[ur, "\underline{DN}"] \arrow[dr, "\underline{AT}"'] & \\ & \mathrm{attr} \end{tikzcd}$$

Fig. 5.2 General diagram valid for all types of entities, where n is the unique name

At this point of the discussion the denotation (den) cannot be further specified since its structure and significance depend on the kind of declaration which created the entity.

The attribute, attr, is a copy of the attribute from the declaration which created the entity. The meaning of the global identifiers in this copy (because of the insertion of unique names) remains fixed for any interpretation and is the meaning given during interpretation of the respective declaration part. This is important since the attribute may contain expressions which are evaluated outside the scope of their global variables (e.g., bounds of controlled array variables).

5.3 PROPER VARIABLES

The following abbreviations and metavariables are used:

gen	generation
eva	evaluated aggregate attribute
mi	mapping information
vr	value representation
pp	pointer part

The general diagram of a proper variable is:

$$\begin{tikzcd}
 & b \arrow[r, "\underline{AG}"] & \langle gen_k, gen_{k-1}, \ldots, gen_1 \rangle \\
n \arrow[ur, "\underline{DN}"] \arrow[dr, "\underline{AT}"'] & & \\
 & attr &
\end{tikzcd}$$

Fig. 5.3a Proper variables

$$\begin{tikzcd}
 & gen_j = (eva_j, mi_j, pp_j) \arrow[dl, "\underline{S}"'] \arrow[d, "\underline{S}"] \arrow[dr, "\underline{S}"] & & \\
vr_1 & vr_2 \quad \cdots\cdots & vr_m
\end{tikzcd}$$

Fig. 5.3b Generations of variables

The Figures 5.3a and 5.3b are valid for all kinds of proper variables and may therefore be taken to represent the general concept of proper variables in PL/I. Without reference to the specific types of proper variables the following general rules can be stated.

General rules:

(1) <u>To allocate</u> a variable means, with respect to Fig. 5.3,[1] to create a generation (gen_{k+1} and to add the generation as the head of the generation list in <u>AG</u>.

The new generation list is then:

$\langle gen_{k+1}, gen_k, gen_{k-1}, \ldots, gen_1 \rangle$

The creation of a generation usually involves the evaluation of aggregate attributes.

(2) <u>To free</u> a variable means, with respect to Fig. 5.3,[1] to delete the head from the generation list. The new generation list is then:

$\langle gen_{k-1}, \ldots, gen_1 \rangle$

---

1) There is also a change in an allocation state (main storage for proper variables) but this allocation state is not part of the diagram for proper variables.

(3) The current generation is the head of the generation list (i.e., gen_k).

(4) An assignment to or initialization of a variable changes or sets the value representations of the current generation.

(5) Any attempt to free gen_1 by a free statement is an error.[1]

(6) eva of any generation is produced upon allocation.

The following rules will distinguish the special types of proper variables as special cases of the general diagram.

The different types of variables:

(1) Controlled variables:

(a) The aggregate name b is created during the prepass and substituted into the declaration.

(b) The generation list is initially set to $<gen_1>$ and updated by the execution of explicit allocate and free statements. The initial generation gen_1 is essentially a null generation containing an attribute part but no pointer part.

(c) eva is either taken from the previous generation or evaluated from attr or from attributes which occur in the allocate statement.

(2) Static variables:

(a) For the aggregate name see (1) (a) above.

(b) A generation is created by the prepass according to the corresponding declaration; the generation list is set to <gen,*> and remains constant during the entire interpretation of the program.

(3) Automatic variables:

(a) A unique aggregate name b is created when the declaration is interpreted (block entry).

(b) A generation gen is created upon the interpretation of the declaration (block entry) and the generation list is set to <gen> and remains constant during the entire corresponding block activation.

(4) External variables:

External variables are either static or controlled. The same unique aggregate name b is substituted into all declarations of the same identifier during the prepass.

---

1) This is part of a mechanism which guarantees that no task frees variables allocated by its mother.

(5) Internal variables:

Automatic variables are always internal and have been dealt with in (3).

For static and controlled variables different aggregate names are created for each declaration during the prepass.

(6) Parameters:

The following is a summary of the possibilities for passing a proper variable as an argument to a parameter.

(a) If the argument is controlled and the parameter is also controlled the aggregate name b is passed to the parameter which therefore shares the generation list with the parameter.

(b) If the attribute of the argument and the corresponding entry declaration match in a certain way, the current generation (or a subgeneration thereof) is passed to the parameter. A new unique aggregate name b is created for the parameter. The parameter therefore shares values with the argument.

(c) In all other cases a new variable (dummy variable) is created and identified with the parameter whose initial value is the value (or part of the value) of the argument. Therefore there is no sharing at all between the parameter and the argument.

The following are notes on some sharing patterns which might occur between two different variables.

Some sharing patterns:

(1) Two different variables have the same aggregate name and therefore share the generation list.

$$\begin{tikzcd}
 & b \arrow[rrdd, "\underline{AG}"] & & \\
n_1 \arrow[ru, "\underline{DN}"] \arrow[rd, "\underline{AT}"'] & & & \\
 & attr & & \langle gen_k, \cdots, gen_1 \rangle \\
 & b \arrow[rru, "\underline{AG}"] & & \\
n_2 \arrow[ru, "\underline{DN}"] \arrow[rd, "\underline{AT}"'] & & & \\
 & attr' & &
\end{tikzcd}$$

Fig. 5.4

This sharing pattern occurs in the following situations:

(a) two external variables having the same identifier;

(b) controlled or static variables created by the same declaration;

(c) a controlled variable (n_1) passed to a parameter (n_2).

(2) Two variables pointing through their generations to non-independent storage and therefore sharing values.

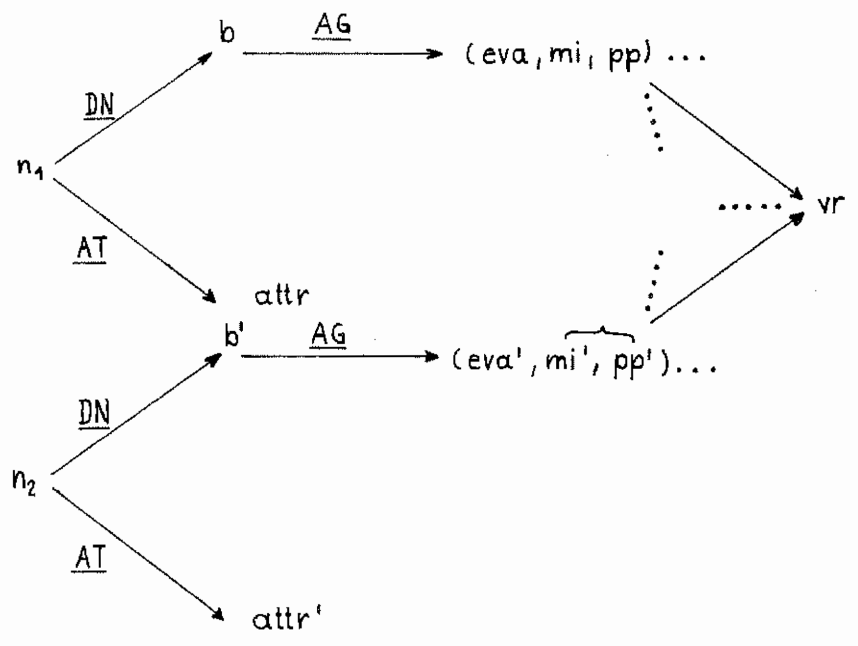


Fig. 5.5

This sharing pattern occurs in the cases (a), (b), and (c) of (1) and in the situation where the generation or a subgeneration of a proper variable is passed to a parameter (case (b) of parameter passing).

5.4 BASED AND DEFINED VARIABLES

(1) Based variables (cf. 10.2.5.3)

The general diagram for based variables is:

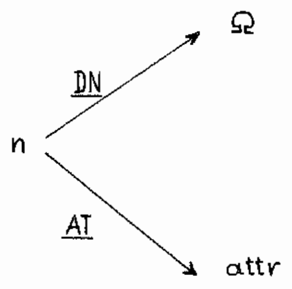


Fig. 5.6

Upon reference the attributes are evaluated and a generation is temporarily created from these evaluated attributes and the pointer given by the pointer qualification of the reference.

(2) Defined variables (cf. 10.2.5.2)

The general diagram is:

$$\begin{tikzcd} & eva \\ n \arrow[ur, "\underline{DN}"] \arrow[dr, "\underline{AT}"'] & \\ & attr \end{tikzcd}$$

Fig. 5.7

The denotation of a defined variable is the evaluated aggregate attribute. The evaluation is done upon interpretation of the respective declaration (prologue). The base, which is contained in the attribute, is evaluated upon reference.

Upon reference a generation is temporarily created from the eva and the evaluation of the base which contributes essentialy a pointer. No sharing patterns and no parameter passing is to be considered for based and defined variables.

5.5 FILES

The following abbreviations are used in this section:

fa	file attributes
f	file name
fd-ea,ea	evaluated environment attribute
id	file identifier
u	file union name
own-inh	own or inherited file
fd-tmt,tmt	transmission error flag
csa	complete set of attributes
ds	data set
FD,fd-	file directory
FU	file union directory
ES	external storage
buf	buffer information
io-ev	attached I/O-events
tn	locked tasks
env	environment attribute

The general diagram for a file is:

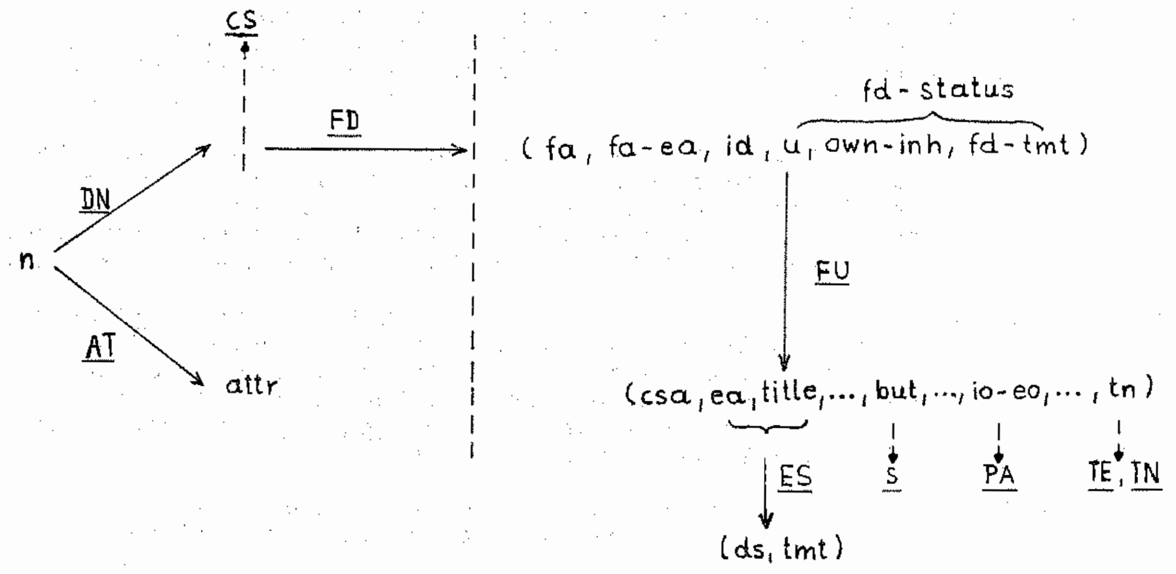

CS
FD
DN
n
AT
attr
fd-status
(fa, fa-ea, id, u, own-inh, fd-tmt)
FU
(csa, ea, title, ..., but, ..., io-eo, ..., tn)
ES
S
PA
TE, TN
(ds, tmt)

Fig. 5.8

A file is the information which is accessible by the unique name n through the state components AT, DN, FD, FU, and ES.[1] The various entries are created or modified at the following stages of the computation: prepass, block prologue, attaching and termination of tasks, data transmission including opening and closing of the file. The entries to the right of the dotted line in Fig. 5.8 are available only at points when data transmission may take place, i.e., when the file has been opened and has not yet been closed.

General rules:

(1) The file name f is created during the prepass, and is unique for each declaration of internal file constants and for all declarations of the same file identifier in the case of external file constants. The file name is substituted into the respective declaration during the prepass. The declared environment attribute is evaluated and is entered into the file directory of the main task under the file name f (fd-ea) together with the file attributes fa and the file identifier id.[2]

Opening of a file amends the entry in the file directory (fd-status in Fig. 5.8) and makes the entry in the file union directory (the file

---

1) The casual access to other state components is represented by dashed arrows in Fig. 5.8.

2) Note: The value of a file variable is a representation of the unique name n of the respective file constant. Therefore n is referred to a file value in the chapters 4. and 12.

union) under a new file union name u which characterizes a "generation" of the file.

(3) Data transmission over the file goes to or from the data set ds which resides in external storage ES. Various components of the file union keep track of the data transmission. In addition, transmission errors are recorded in tmt and fd-tmt (Fig. 5.8).

(4) Closing of a file deletes all entries made at opening, i.e., the fd-status and the file union. Assuming that the file union name u has been stored somewhere, it is no more possible to localize ds in ES on the basis of the file union name. This remains true even if a following opening establishes another generation of the file.

(5) Attaching of a task provides the new task with a copy of FD such that all components fa, fd-ea, and id are exact copies. From the fd-status of all open files only the file union name is copied. The components own-inh will be empty for the life time of the attached task (this is the indication for inherited files). The components fd-tmt will be initially empty.

(6) Termination of a task causes closing of all files opened by this task which have not been closed so far (closing of all owned files).

There are no pecularities of argument passing since parameters can only be file variables.

A more detailed description of file directories (and the modelling of standard system print files) is given in section 12.2.1. Section 4.3 describes the selection of a particular data set in ES, and the structuring imposed on the data set by the characteristics of the file.

Sharing Patterns

(1) Files sharing the on-condition actions:

files having the same file name, i.e.,

(a) external file constants having the same identifier

(b) file constants created by the same declaration.

(2) Files sharing the file union:

files having the same file union name, i.e., file inherited by task calls.

As a consequence situation (1) applies.

(3) Files sharing the data set:

There is a function which yields for any ea and title a datasetname by which the data set can be retrieved from ES.

Consequently, files having the same ea and title in the file union share the data set. This is, however, only a sufficient but not a necessary condition. A more detailed description of the subject may be found in 4.3.1.

5.6 PROCEDURES

The following abbreviations are used in this section:

ba	block activation name
bpp	block prefix part
param-list	parameter description list
ret-type	return type

The general diagram for a procedure is:

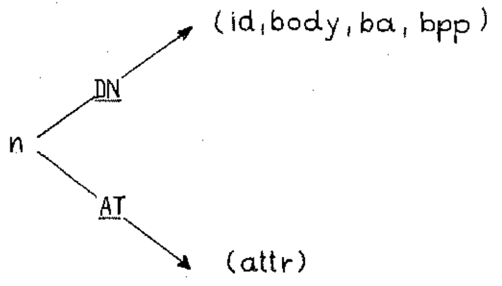


Fig. 5.9

The entire entity is created upon block entry and remains unchanged during execution (in that sense procedures may be considered to be names of constants).

The identifier points to the statement with which the interpretation of the body has to start.

The body is essentially the text to be interpreted when the procedure is called. The body is defined in the corresponding declaration part in case of internal procedures. For external procedures a unique name is found in the declaration which allows the retrieval of the body of the external procedure. The abstract structure of the body has been given in 2.1.3.

The block activation name is the name of the activation in which the corresponding declaration was interpreted.

The block prefix part is relevant for condition enabling.

There are no interesting sharing patterns to be discussed for procedures, since the entire diagram remains constant during interpretation.

5.7 GENERIC NAMES

The general diagram is:

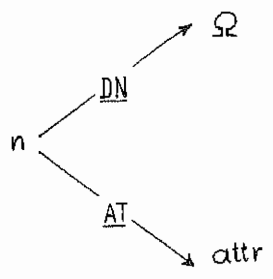


Fig. 5.10

The denotation is Ω.

The attributes are a list of pairs (ref,descr-list) where each pair consists of an entry reference and a parameter description list. The reference of each pair refers to a procedure. Upon reference to a generic name a specific pair is selected by comparing the argument list of the reference with the various parameter description lists of the list. The procedure referenced by the reference of the selected pair is then called.

There is no parameter passing to be considered since there are no generic parameters.[1]

5.8 BUILTIN FUNCTIONS

The general diagram for builtin functions is:

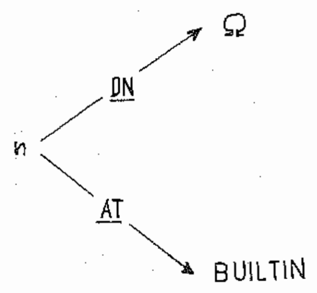


Fig. 5.11

The declared identifier id determines uniquely the builtin function to be evaluated. The definition of the builtin function is contained in the interpreter (and not given by standard declarations).

---

1) If an argument is a generic reference then the result of th e generic selection is passed, i.e., a procedure.

5.9 LABELS

5.9.1 LABELS WHICH SERVE AS DESIGNATION OF GOTO STATEMENTS

The general diagram is:

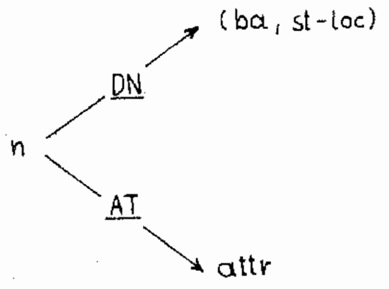


Fig. 5.12

The denotation is a pair which consists of a block activation name and an index list which identifies the statement location. The unique name ba determines the block activation and the statement location determines the statement within this block activation to which control is passed in case of a goto statement refering to that label. No argument passing is to be considered. A dummy label variable is always created and passed to the parameter, when a label occurs as argument.

5.9.2 FORMAT LABELS

The general diagram is:

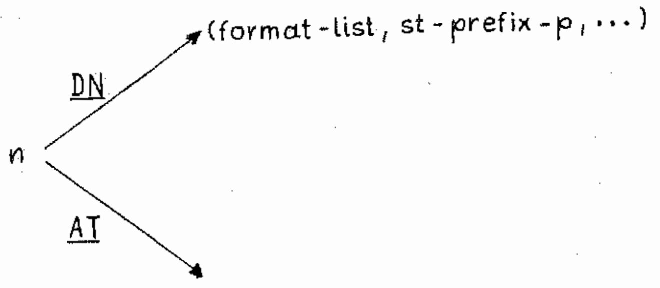


Fig. 5.13

The format-list is taken from the labeled statement. The environment is that which determines the interpretation of the format-list. The st-prefix-p is constructed from the relevant part of the statement updated when the declaration is interpreted. The additional components serve checking purposes.

5.10 ATTENTIONS

The following abbreviations and metavariables are used:

EV	attention environment directory
AN	attention directory
ea	evaluated environment
attn	attention

The general diagram is:

$$\begin{tikzcd}
 & b \arrow[r, "EV"] & \left.\begin{matrix} ea \\ id \end{matrix}\right\} \arrow[r, "AN"] & attn \\
n \arrow[ur, "DN"] \arrow[dr] & & & \\
 & attr & &
\end{tikzcd}$$

Fig. 5.14

A unique name b is created during the prepass for each attention identifier and inserted into the respective declaration (similar to external). The prepass also evaluates the environment attribute and makes the appropriate entry into EV. There is a function which yield for any identifier and evaluated environment attribute a name which gives access to an entry in AN. The entry in AN is initially made by the interpretation of an enable statement. The attention attn can be changed by attention occurrences enable statements, disable statements, asynchroneous attention interrupts and access-statements.

5.11 SOME REMARKS

After having enumerated all types of names (except condition names) which can be declared in a PL/I program, one may ask for which types of names can the associated diagram change dynamically. The names for which the diagram may change will be called (in this section) variables; the remainder constants.

variables:

(1) proper variables
(2) files

constants:

(1) based and defined variables
(2) procedures
(3) generic names
(4) builtin functions
(5) labels

The study of sharing patterns is only relevant for variables and not for constants, since these patterns express whether updating a part of an entity means automatically updating of a part of another entity. Consider the example given in Fig. 5.15a where both n_1 and n_2 have the <u>same</u> y as a component. Since each name has its own copy of y, updating of y of one name would only mean updating of its own copy of y. If, however, there is only one copy of y owned by both names as indicated in Fig. 5.15b then any updating of y via one name would also mean updating of y for the other name. In the latter case only it is said that n_1 and n_2 <u>share</u> y.

$$\begin{tikzcd}
n_1 \arrow[d] & n_2 \arrow[d] \\
y & y
\end{tikzcd}$$

Fig. 5.15a n_1 and n_2 have the same y

$$\begin{tikzcd}
n_1 \arrow[d] & & n_2 \arrow[d] \\
x \arrow[dr] & & x \arrow[dl] \\
& y &
\end{tikzcd}$$

Fig. 5.15b n_1 and n_2 share y

Chapter 5 has so far not considered tasking. In the sequel the relation of tasking to the general diagrams will be discussed briefly. of tasking to the general diagrams will be briefly discussed. There are certain components in the state of the PL/I machine, called task global, which are shared by all active tasks and there are other components, called task local, which are privat to a specific task, i.e., not shared among different tasks. There are two task local state components, AG and FD, with respect to the state components mentioned in the general diagrams. All other state components mentioned in the diagrams are task global. When a task is attached a modified copy of the AG and FD of the attaching task is made for it.

The modification of the copy of AG consists in deleting all generations from the generation lists except the current ones. Let AG_1 be an aggregate directory and let AG_2 be the modified copy made from AG_1 for a task to be attached.

The diagrams in Fig. 5.16a,b show the versions of a proper variable for the two tasks.

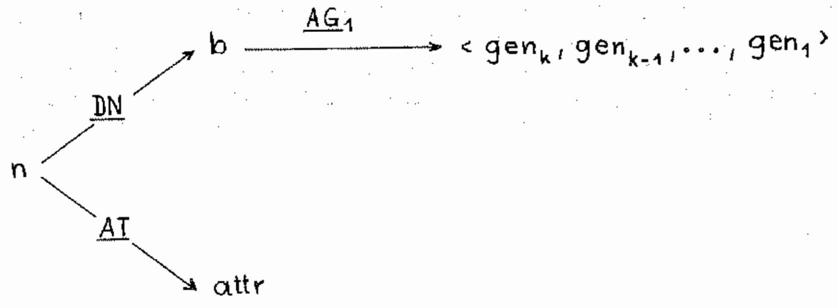


Fig. 5.16a attaching task

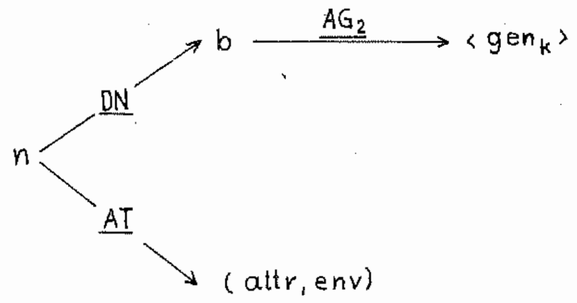


Fig. 5.16b attached task

The two versions of the variable obviously share storage via gen_k. They have, however, their own copies of the generation list and will therefore not share storage via generations allocated after the task is attached. The rule that gen_1 of a generation list must not be freed guarantees that the daughter task will not free generations allocated by the mother task.

The modification of the copy of FD consists in changing all occurrences of * (own) to Ω (inherited). Since any opening creates an entry (u,*,...), the interpreter can always test whether a file was opened by the current or some mother task. Let FD_1 be a file directory and let FD_2 be the modified copy made from FD_1 for a task to be attached. The diagrams 5.17a,b show the versions of a file for the two tasks.

$$\begin{tikzcd}
 & f \arrow[r, "\underline{FD_1}"] & (u,*,\dots) \arrow[dd, "\underline{FU}"] \\
n \arrow[ur, "\underline{DN}"] \arrow[dr, "\underline{AT}"'] & & \\
 & attr & (csa,title,\dots) \arrow[d, "\underline{ES}"] \\
 & & (ds,tmt)
\end{tikzcd}$$

Fig. 5.17a attaching task

$$\begin{tikzcd}
 & f \arrow[r, "\underline{FD_2}"] & (u,INH,\dots) \arrow[dd, "\underline{FU}"] \\
n \arrow[ur, "\underline{DN}"] \arrow[dr, "\underline{AT}"'] & & \\
 & attr & (csa,title,\dots) \arrow[d, "\underline{ES}"] \\
 & & (ds,tmt)
\end{tikzcd}$$

Fig. 5.17b Attached task

The two versions obviously share the information in FU and the data set. If, however, the mother closes the file and opens another file with this unique name, the two versions will no longer share information, via the file union name, since the mother will create a new file union name upon opening.

6. THE COMPUTATION OF THE PL/I MACHINE

Corresponding section of /5/:

3.7 The computation of the PL/I machine

6.1 THE INITIAL STATE OF THE PL/I MACHINE

The interpretation of a program starts with an initial state ξ_0 which essentially is a cleared machine (cf. 4.1 of /5/). It contains one active task, the main task. The only components which are not cleared and which essentially determine the computation are the following:

(1) The external storage. It contains, in particular, the input data for the computation.

(2) The main storage. The initial state of the main storage may influence a computation, though a well written program generally should eliminate this influence (reference to a variable for which storage is allocated but not initialized depends on the storage before allocation).

(3) The control of the main task. It contains only the instruction <u>int-program</u>(t,call,gen) which is executed as first instruction and initiates the complete program interpretation. The three arguments of this instruction are

  (1) the program t to be interpreted as described in chapter 2,

  (2) a call statement or function reference call, specifying the entry point at which the program interpretation is to be started and possibly arguments to be passed to the parameters of the entry point (cf. 2.2). The concrete specification of this call statement or function reference is implementation dependent, e.g., by control cards or (in the F implementation) by a procedure option MAIN included in the concrete program itself;

  (3) optionally a generation gen to which the returned value is to be assigned if the program is activated by a function reference.

The initial instruction handles the program t similar to the interpretation of a begin block (cf. 8.2), but instead of the statement list of a block the initial call statement or function reference is interpreted. Moreover, before the bodies for the declared external entry identifiers are entered as parts of denotations into the denotation directory <u>DN</u>, their text is modified by the so-called prepass.

6.2 THE STATE TRANSITIONS

The transition from a state of the PL/I machine to its successor state occurs in three steps, which are discussed in the following.

(1) The computation step. This step is controlled by the instructions contained in the control parts of the various tasks. First, one of the active tasks is selected for execution. This is done by an implementation-defined function, which returns the name of a specific task. This task name is inserted in the TN-component of the state, specifying the current task. The task name permits the access to the state components local to this task in PA (cf. 3). From the control part of this task, one of the instructions is selected which are candidates for execution.

The control part is a tree-like collection of instructions, where the instructions located at the terminal nodes of the tree are those which are candidates for execution. There is, consequently, a certain freedom as to how to proceed in the computation. This freedom accounts for the fact that in some places in PL/I the sequence of certain actions is left unspecified (e.g., the order of evaluation of operands in an expression).

Each instruction defines a specific state transformation. The state transformation defined by the instruction selected for execution is actually performed.

(2) The environment step. There are certain changes in the state possible which are not controlled by the program being executed. The state obtained by the computation step can be modified by certain permissive changes in the internal or external storage effected by the environment of the machine, by an updating of the time component, and by incoming attentions and reply messages.

(3) The interrupt step. Tests are made, whether the normal computation has to be interrupted because of changes made in the environment step. In this step tasks which are in the wait state may be activated (because of incoming reply messages, input to transient files, incoming asynchronous attentions, or if the time condition for a delay statement is satisfied), tasks receiving asynchronous attentions are also interrupted in their normal flow of execution to enforce the call of an associated on-unit.

The above three steps define the next state in the computation. If this is not an end state, again a next state is produced according to the same rules.

7. TASKS

Corresponding sections of /5/:

5. Tasks

3.1 Parallel actions

The following abbreviations are used in this chapter:

AG	aggregate directory
BA	block activation name
c,C	control
CI	control information
CS	condition status
D	dump
EI	epilogue information
EN	attention enabling state
ev	event variable
FD	file directory
io-ev	input-output event
pa,PA	parallel action part
pri	priority
s,S	storage
TD	time and date part
te,TE	task-event specification
tn,TN	task-event name
tv	task variable

This chapter describes the parallel execution of parts of a PL/I program and the synchronization of such parallel executions.

In a PL/I program, it is possible to specify in a call statement that the called procedure body is to be executed in parallel with the calling block; i.e., the calling block continues with the execution of the statements following the call statement while the called procedure body is executed. Each parallel execution of a procedure body is called a _task_. Each task may itself call other tasks. The execution of the procedure body by which the program is started is itself a task, the so-called _main task_. A task is called _active_ from the time when it is started up to the time when it has terminated its last actions.

Furthermore, it is possible to specify in an input or output statement that the data transmission is performed in parallel with the execution of the task which contains the statement; i.e., the task continues with the execution of the statements following the input or output statement while the data transmission takes place. This parallel data transmission is called an _I/O-event_.

Usually a task, which started a new task or an I/O-event, has to make use of the effects of that task or I/O-event at some later time. To be sure that these effects have been completed when they are needed, there are means for _synchronization_ of tasks and I/O-events. The synchronization is performed by _event variables_, which are set either explicitly by assignment statements or automatically on completion of a task or I/O-event. They are inspected by a wait statement which delays the execution of its task until specified event variables are set.

The following sections describe the realization of these features of the language by the formal model of the PL/I machine. To be concise, usually only tasks are mentioned, though most of the discussions are valid also for I/O-events.

7.1 PARALLEL EXECUTION

Whenever the language specifies that tasks are to be executed "in parallel", a concrete implementation, depending on its hardware environment, may choose one of the following alternatives: Either it may execute them really simultaneously, e.g., using different processing units. Or it may execute one task after another one. Or it may execute them "intermixed", e.g., performing first some actions of one task, then some actions of another one, then continuing the first task, and so on. The only restriction is that no actions of tasks which have to wait for actions of other tasks (cf. 7.5) are to be performed. The choice, which tasks are to be executed first, may be influenced by priorities specified in the program, though this influence is not defined by the language.

In the formal definition, the parallel execution of tasks is modelled by a sequential machine: An instruction of one task is executed after another instruction of possibly another task. In each state of the computation, an implementation defined function, the _priority scheduler_, determines out of which task an instruction is to be executed next. It does not determine a task which is in a wait state (cf. 7.5).

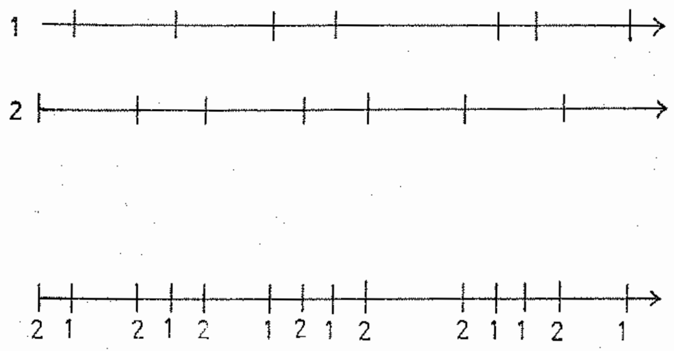


Fig. 7.1 Sequentializing model of instruction executions of two parallel tasks

This sequentialized model describes the language correctly (i.e., its possible computations are equivalent to the different implementations permitted) as long as the executions of instructions of different tasks do not influence each other. Mutual influencing of instructions of different tasks occurs by changing and accessing of common state components, mainly by use of the same piece of storage or external storage.

When using the same piece of storage or external storage, the question of <u>uninterruptable actions</u> occurs. Principally, in the formal model the execution of an instruction, which transforms one state into its successor state, is understood as uninterruptable. Nevertheless, in a concrete implementation one instruction execution of the formal model may be realized by a series of elementary, uninterruptable, actions. Or different instruction executions of the formal model may be realized by a single elementary, uninterruptable, action. Thus, it may happen that the simultaneous or sequentially mixed execution of different tasks may lead to results which could not occur in any computation of the formal model.

<u>Example</u>:

```
A:BEGIN; DCL X CHAR(3) INIT('ABC'), Y CHAR(3);

         CALL B TASK;
         ...
         Y = X;
         ...
B:PROC;
         ...
         X = 'XYZ';
         ...
```

In the formal model the assignment X='XYZ' in task B and the reference to X in task A are performed both by single instructions. That means, that X='XYZ' is executed either before or after Y=X and thus finally the value of Y is either 'ABC' or 'XYZ'. In a concrete implementation the assigning and referencing might be performed character-wise. Then it might occur that X is referenced in task A after the first character had been assigned in task B, i.e., finally the value of Y might be 'XBC' or 'XYC' or even something else.

For these reasons the language says that the result of a program assigning and referencing the same piece of storage or external storage by different tasks is undefined, if these tasks are not synchronized appropriately to avoid such simultaneous access. This undefinedness is not expressed by the formal model (it will always yield a set of well defined results, as in the above example). Apart from this undefinedness, the model reflects all situations which are allowed by the language and which may occur in concrete implementations.

To have the possibility of synchronizing tasks in a defined way, however, certain actions are defined to be uninterruptable by the language. These actions are the creation and termination of a task or I/O-event and the changing and accessing of event variables.

7.2 REPRESENTATION OF TASKS IN THE STATE OF THE PL/I MACHINE.

Corresponding sections of /5/:

3.1.1 The parallel action part PA

3.1.2 The current task event name TN

3.1.3 The task-event specification TE

Certain state components, e.g., the storage S, are common to all tasks. They are called the global state components and serve (among other purposes) for communication between tasks. The other state components, e.g., the control C, are owned by the single tasks, i.e., each task has its individual ones. They are called the task local state components and carry all information needed within the single tasks. Usually no task uses the task local state components of other tasks; exceptions from this rule are the creation of new tasks, the abnormal termination of other tasks and inspection of information about event variables, i.e., situations concerned with explicit synchronization between tasks.

The global state components are immediate components of the state ξ of the PL/I machine, while all task local state components are subcomponents of one immediate component of the state ξ, namely of the parallel action part PA = s-pa(ξ).

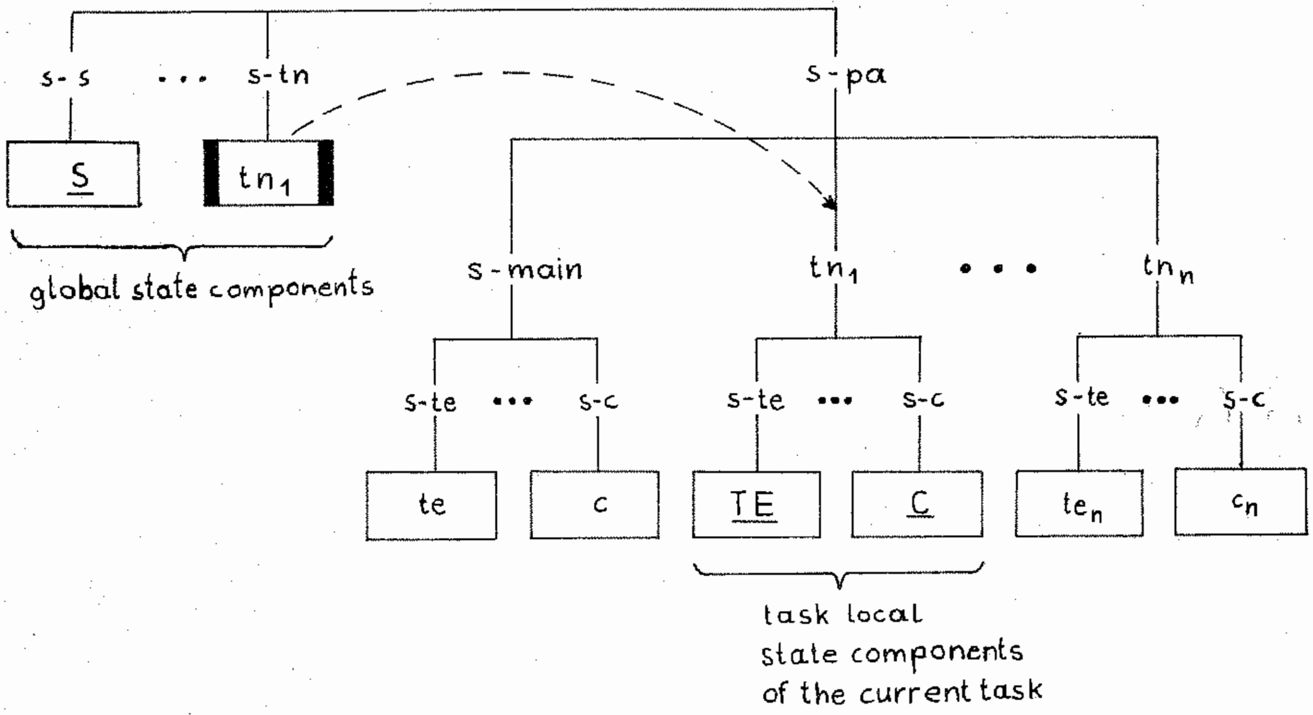


Fig. 7.2 Tasks in the state of the PL/I machine

The parallel action part PA contains for each active task an immediate component, selected by a unique selector, called task-event name tn. The task-event names uniquely identify the single tasks (and I/O-events). The task-event name of the main task is the selector s-main, those of all other tasks are unique names created immediately before the tasks are started. At each state of the machine, the priority scheduler (cf 7.1) determines the task-event name tn of that task from which an instruction is to be executed; this task-event name is entered into a global state component, the current task-event name TN=s-tn(ξ). By means of this component TN, one has access to that component of the parallel action part PA which is associated with the task currently executed. This task is called the current task.

The task local state components of each active task constitute the component of the parallel action part PA selected by the task-event name of that task. So, the parallel action part PA contains the task local state components of all active tasks, combined into one component for each task. E.g., the control of the task identified by the task-event name tn is the s-c component of the component selected by tn from PA, i.e., s-c•tn•s-pa(ξ).

In the initial state ξ_0 of the machine, there is only one active task, the main task. Therefore the current task-event name TN of the initial state is the selector s-main and the parallel action part PA of the initial state has only one component, selected by s-main.

In each state, the task local state components of the current task are called the current task local state components, e.g., the current control. They are denoted by underlined capital letters, like the global state components. They are found by applying the current task-event name <u>TN</u>=s-tn(ξ) to <u>PA</u>, e.g., <u>C</u>=s-c•<u>TN</u>(<u>PA</u>). A reader, who is not interested in tasking questions, may assume <u>TN</u> to be constant (the selector s-main). He may speak e.g., of "the control <u>C</u>" instead of "the current control <u>C</u>" and ignore the fact that the selectors for the task local state components, e.g., s-c•<u>TN</u>•s-pa, for the control <u>C</u> are more complicated than those for the global ones, e.g., s-s for the storage <u>S</u>. Also throughout the present document the current task local state components are named as, e.g., "the control" instead of the current control, etc., whenever only one single task is under consideration.

It should be remembered that the parallel action part <u>PA</u> contains besides the entries for the active tasks also similar entries for the active I/O-events (cf. 12.5.1) and entries for attention events (cf. 11.2.2.1).

There is one special task local state component which, for each single task, carries information about its status. This component is the task-event specification <u>TE</u>.

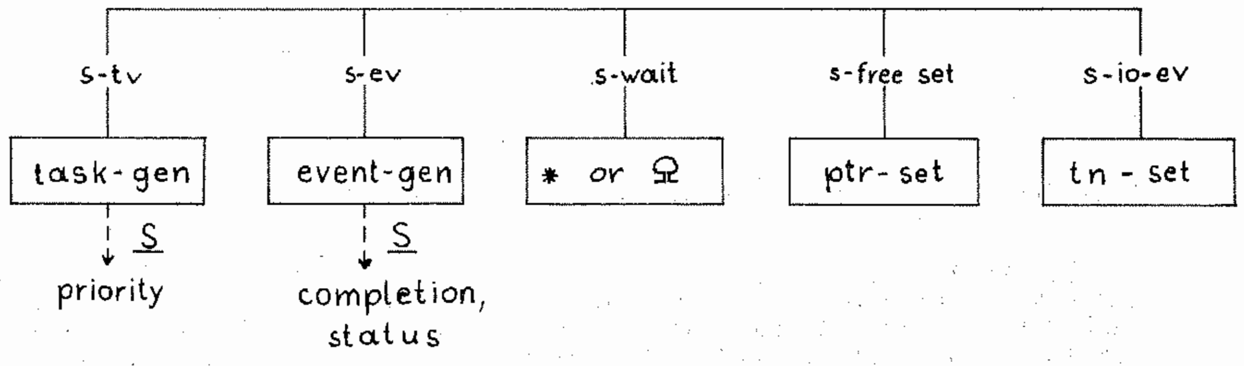


Fig. 7.3 Task-event specification <u>TE</u> of a task

It consists of the following five components:

(1) The generation of the <u>associated task variable</u>. This variable is either specified by the call statement which initiated the task, or a dummy variable is created. The value of this variable (to be found in the part of the storage <u>S</u> belonging to the generation) is an integer, the <u>priority</u> of the task. The priorities of all active tasks are considered (in an implementation defined way) by the priority scheduler (cf. 7.1) to determine which is the current task for the next instruction execution. When a task is started its priority is determined in one of the following three ways:

(a) The initiating call statement specifies a relative priority. Then this relative priority is added to the current priority of the calling task, the resulting value is taken as priority of the called task and assigned to its task variable.

(b) The call statement specifies no relative priority but a task variable. Then the value of the task variable is left unchanged and taken as priority of the called task.

(c) The call statement specifies neither a relative priority nor a task variable. Then the current priority of the calling task is taken as priority also of the called task and assigned to its task variable.

The value of the task variable, and thereby the priority of the task, may be changed at any time by assignment of a new value, with the only exception that as long as the task is active, this is possible only by use of the priority pseudo variable. No task variable can be associated with two tasks at the same time.

(2) The generation of the <u>associated event variable</u>. This variable is either specified by the call statement which initiated the task, or a dummy variable is created. It is used for synchronization of other tasks with the termination of this task (cf. 7.5). When a task is started the completion value of its associated event variable is set to "incomplete" (0-BIT); it cannot be changed as long as the task is active and is set to "complete" (1-BIT) automatically at the end of the task. As long as a task is active only the status value of its event variable can be changed (by use of the status pseudo variable). No event variable can be associated with two tasks at the same time.

(3) The <u>wait state flag</u>. It is usually empty. Only if the task is waiting for synchronization with some effect of other tasks, this component is present. The priority scheduler (cf. 7.1) chooses only such tasks for execution whose wait state flags presently are empty. The flag is set by the task itself whenever it comes into a situation where it has to wait for some other action. The wait flags of <u>all</u> tasks are deleted whenever an action occurs for which possibly a task might be waiting. Thereby all waiting tasks are reactivated, and each of them may determine whether it can continue now or whether it has to wait further, i.e., to reset its wait state flag.

(4) The <u>based free set</u>. All storage allocated by a task by based allocation has to be freed by the task, at latest at its termination. To preserve the information necessary for this purpose, each based allocation enters the pointer of the allocated storage into the based free set of the task-event specification <u>TE</u> of the current task, and each based freeing deletes the pointer from this set. At the start of a task this set is empty, at the end of a task the storage of all pointers left in the set is freed automatically.

(5) The <u>I/O-event set</u>. All I/O-events started by a task, which are yet running, are to be terminated when the task terminates. For this purpose, similarly as with the based storage, at each start of an I/O-event its task-event name is entered into the I/O-event set of the task-event specification <u>TE</u> of the current task. At the start of a task this set is empty, at the end of a task all I/O-events, whose task-event names are entered into the set and which are yet active, are terminated.

7.3 ATTACHING OF TASKS

Corresponding section of /5/:

5.1 Attaching of tasks

The creation of a new task is called attaching a task. The task which attached the new task, by execution of a call statement with a task option, is called the mother-task, the newly attached task the daughter task.

A task option of a call statement, which causes the attaching of a new task rather than just the establishing of a new block activation within the current task, consists of three components:

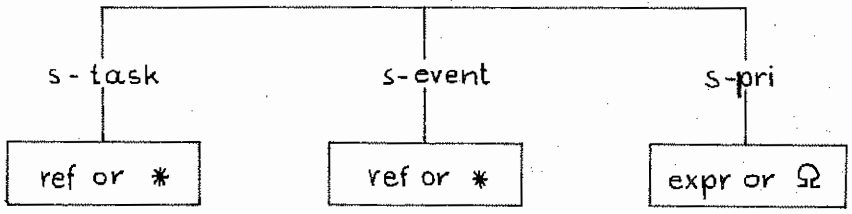


Fig. 7.4 Task option of a call statement

(1) a reference for the task variable of the daughter task, or an asterisk if a dummy task variable is to be created;

(2) a reference for the event variable of the daughter task, or an asterisk if a dummy task variable is to be created;

(3) an expression specifying the priority of the daughter task relative to that of the mother task, or empty if the priority of the daughter task is to be taken either from the specified task variable or from that of the mother task.

When a new task is to be attached, the mother task creates a new unique name tn and enters a new component for the daughter task into the parallel action part PA, using the created unique name tn as task-event name of the daughter task. The new component of PA consists of the following task local state components for the daughter task:

(1) The task event specification TE of the daughter task is constructed mainly from the components of the task option of the call statement. The task and event variables are initialized by priority and completion values as described in 7.2.

(2) The aggregate directory AG of the daughter task is constructed from that of the mother task by taking over only the head generations of all entries. By this mechanism it is guaranteed that both mother and daughter tasks can use the current generations of all variables, that both can continue the generation stack of controlled variables independently by

allocations and that each task can free only those generations which it had allocated itself (no task frees the last generations in the generation lists in its own aggregate directory).

(3) The file directory FD of the daughter task is constructed from that of the mother task by taking over all its entries, but deleting all s-own components. By this mechanism all existing entries are denoted as inherited and it is guaranteed that the daughter task may use all files opened by the mother task before the attaching, that both mother and daughter task may open independently further files, and that each task can close only those files which it had opened itself (no file closes files which are not denoted as "own" in its own file directory).

(4) The attention enabling state EN of the daughter task contains no enabled, no associated and no waiting attentions.

(5) The dump D and the condition status CS of the daughter task are copied from the current ones of the mother task. Thereby it is guaranteed that the rules of the dynamic block structure of PL/I (cf. 8) apply to the nested block activations, which will be established within the daughter task, in the same way as if they have been nested in the current block activation within the mother task. I.e., the dynamic structure of nested block activations will be continued idependently in the mother task and in the daughter task. This applies especially to the scope rules for the meaning of identifiers and to the inheritance rules for condition enabling and condition actions. This continuation of the sequence of nested block activations into the daughter task is the reason, why no block is terminated before all tasks attached by it have been terminated (cf. 7.4); otherwise the sequence of nested block activations would be interrupted, the daughter task would still use identifiers inherited from the mother task which are already obsolete.

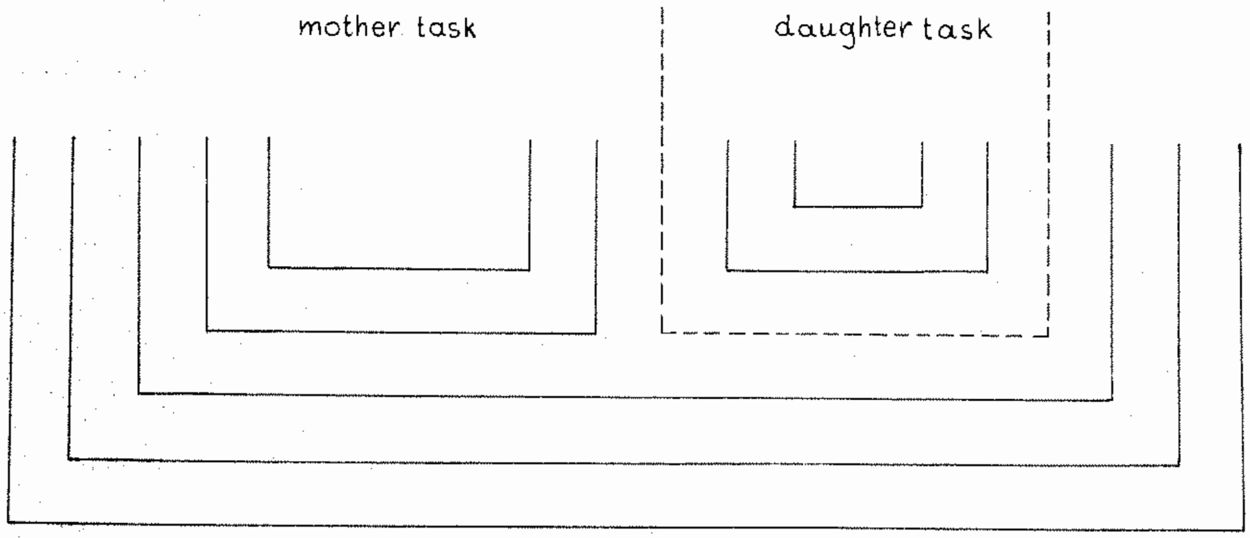


Fig. 7.5 Structure of dynamically nested block activations of mother task and daughter task

It should be noted that actually the inherited dump will be used by the daughter task only for two purposes: for reestablishing the condition actions of the surrounding block activation by the revert statement, and for preventing the call of an entry constant, declared in a block activation other than a dynamically surrounding one, by a call statement calling an entry variable.

(6) The block activation name BA, the epilogue information EI and the control information CI of the daughter task are initially empty. An empty block activation name and epilogue information is characteristic for the outermost block activation level of a task. This fact is used by the goto statement, when successively terminating block activations in the dump to prevent a goto into block activations which were established by the mother task.

(7) The control C of the daughter task contains instructions for performing a normal (non-task) procedure call within the daughter task, and after return from that procedure body the termination of the daughter task (cf. 7.4). After the new component for the daughter task has been entered into the parallel action part <u>PA</u> by the mother task, the daughter task is immediately active, has the same rights as all other existing tasks and can execute these instructions in its control.

In addition to the creation of the new component for the daughter task in the parallel action part <u>PA</u> the mother task enters the task event name tn of the attached daughter task into the epilogue information of its current block activation. This is done to keep track of all daughter tasks attached during a block activation, in particular, to terminate all these daughter tasks before the block activation in the mother task terminates (cf. 7.4).

<u>7.4 TERMINATION OF TASKS</u>

Corresponding section of /5/:

5.2 Termination of tasks

A <u>normal termination</u> of a task occurs after normal return from the procedure body called when the task was attached. After this return, all block activations established by the task (except the outermost one) have already been terminated. Then, the task performs the following actions:

(1) closing of all files opened and not yet closed by the task (to be found in the file directory <u>FD</u> of the task);

(2) abnormal termination of all I/O-events started by the task which are not yet terminated (to be found in the I/O-event set of the task-event specification <u>TE</u> belonging to the task);

(3) unlocking of all keys locked by the task;

(4) disabling of all attentions enabled by the task (to be found in the attention enabling state <u>EN</u> of the task);

(5) freeing of all controlled and based storage allocated and not yet freed by the task (to be found in the aggregate directory <u>AG</u> and

the based free set of the task-event specification TE of the task, respectively);

(6) setting the completion value of the event variable associated with the task to "complete";

(7) The last action of the task is to delete its own component from the parallel action part PA; i.e., it removes all its own task local state components from the state. Thereby the representation of the task is completely removed from the state and it is no more active.

An abnormal termination of a task occurs, whenever a task is interrupted during its normal flow by an action causing it to terminate. These interrupting actions are:

(1) execution of an exit statement by the task itself,

(2) execution of a stop statement by any task (terminating the main task),

(3) termination of that block activation in the mother task by which the task was attached.

If a task is to be terminated abnormally the same actions as for a normal task termination are performed, but before that all its active block activations are finished one after the other by execution of the normal block epilogue (cf. 8.2.4). Among other actions, the epilogue of each block activation causes all daughter tasks, attached by the block activation, themselves to terminate abnormally. Final termination of a block activation does not take place before all daughter tasks have in fact completely terminated. Since, in turn all daughter tasks then finish their block activations and thereby terminate all their own daughter tasks and so on, the result is that abnormal termination of a task automatically terminates all descendent tasks before. More precisely, this mechanism ensures that the general principle of the PL/I block concept is obeyed: No block activation is terminated before all its dynamically nested block activations (cf. Fig.7.5) have been terminated, whether they belong to the same task or to any descendent task.

In particular, this mechanism of automatically terminating all descendent tasks when a task is to be terminated, is used by the stop statement. Irrespectively by which task a stop statement is to be executed, it causes the main task, and thereby automatically all other tasks, to terminate abnormally.

7.5 SYNCHRONIZATION OF TASKS

Corresponding sections of /5/:

5.3 The wait statement

3.1.5 The event trace ET

The synchronization of two tasks is performed by means of an event variable (which has to be known to both tasks): it is set by explicit or automatically performed assignment in the one task and inspected by a wait statement in the other task. The value of an event variable consists of two components: the status value (which is not relevant for synchronization) and the completion value. The completion value is a bit denoting either "incomplete" (0-BIT) or "complete" (1-BIT). A wait statement inspects whether specified event variables are "complete", otherwise it waits until they are.

Usually an event variable may be set to "incomplete" or "complete" by explicit assignment of the value of another event variable or by explicit assignment of a bit value by means of the completion pseudo variable. The completion value of an event variable, however, which is associated with a task (cf. 7.2), with an I/O-event (cf. 12.5.1) or with an attention event (cf. 11.2.2.1) cannot be changed by explicit assignment as long as that task or event is active (i.e., as long as the generation of the event variable is entered into the task-event specification TE of any component of the parallel action part PA). Such an event variable is automatically set to "incomplete" when the task or event is attached and to "complete" when it is deleted. Thus, by a wait statement a task can be synchronized either with a specified point (the point of an explicit assignment to "complete") in another task or with the termination of another task, of an I/O-event or of an attention event.

A wait statement recognizes an event variable as complete if this event variable satisfies one of the following three conditions on inspection:

(1) its completion value is in fact "complete", or

(2) its completion value, though it is now possibly "incomplete" again, has been "complete" at some previous time during the current wait statement, or

(3) it is associated with an I/O-event, which:

(a) was attached by the task containing the wait statement, and

(b) is not yet terminated (and therefore the completion value of the event variable is "incomplete"), but

(c) has already finished its data transmission.

Such an I/O-event is called semi-complete.

The first case is recognized simply by inspecting the value of the event variable in its storage. A special global state component, the event trace ET, serves to recognize the second case. It records the order in time of all actions setting event variables to "complete" and all starts of wait statements: It is a list to which, whenever an event variable is set "complete" (by any action of the PL/I machine) the generation of the event variable is added as new last element. Whenever a wait statement starts, it creates a unique name as its individual identification and adds this unique name to the event trace. So at any time, by inspecting the event trace, a wait statement can recognize which event variables have been set complete later than its own start.

<u>Example</u>:

```
        DCL(E1,E2,E3) EVENT;
        COMPL(E1), COMPL(E2), COMPL(E3)='0'B;
        CALL A TASK PRIORITY(3);
        CALL B TASK PRIORITY(2);
        CALL C TASK PRIORITY(1);

      A:PROC;                              ]
            ...                            ]  1
         W2:WAIT(E2);                      ]
            COMPL(E2)='0'B;                ]
         L1:COMPL(E1)='1'B;                ]
            ...                            ]  4
            COMPL(E1)='0'B;                ]
            ...                            ]
         W3:WAIT(E3);                      ]
            ...                            ]  7
            END A;                         ]

      B:PROC;                              ]
            ...                            ]  2
         W1:WAIT(E1);                      ]
            ...                            ]  5
            END B;                         ]

      C:PROC;                              ]
            ...                            ]  3
         L2:COMPL(E2)='1'B;                ]
            ...                            ]  6
         L3:COMPL(E3)='1'B;                ]
            ...                            ]  8
            END C;                         ]
```

Assume that task A has the highest priority and is selected for execution by the priority scheduler whenever possible, that task B has the next priority and is selected for execution whenever A but not B has to wait, and that task C has the lowest priority and is selected only if both A and B have to wait. The interesting wait statement is the one labelled W1 in task B. The machine executes the program sections in the order given by the numbers at the right margin. When the flow of control initially comes to the statement W1 (after the sections 1 and 2), its event variable E1 is "incomplete", thus it has to wait and the sections 3 and 4 are executed. In section 4 the event variable E1 is set "complete" and then "incomplete" again. When (after section 4) the statement W1 inspects E1 again, E1 is "incomplete" but has been "complete" in the meantime, thus task B can continue. The event trace up to this point is the following (denoting the unique names created for the wait statements by W1, W2, W3 and the generations of the event variables by E1, E2, E3):

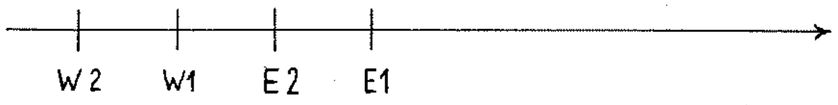


Fig. 7.6 Example of the event trace ET.

A semi-complete I/O-event is characterized by the fact that its component is yet contained in the parallel action part PA, but its control is exhausted. When a wait statement recognizes that one of the event variables it has to wait for is associated with a semi-complete I/O-event, it has fully to complete that I/O-event, i.e., among others to raise I/O-conditions (cf. 12.5.1), to remove the corresponding component from the parallel action part PA and to set the associated event variable to "complete". It should be noted that this completion of a semi-complete I/O-event is performed only by that task which started the I/O-event. Wait statements occurring in other tasks which incidentally wait for the same event variable do not recognize it as semi-complete; they have to wait until it is completed by a wait statement in the right task as described above.

A wait statement has to wait until a given number (the wait count) out of a specified list of event variables (the event list) is recognized as complete, as described above. This is done in the following way:

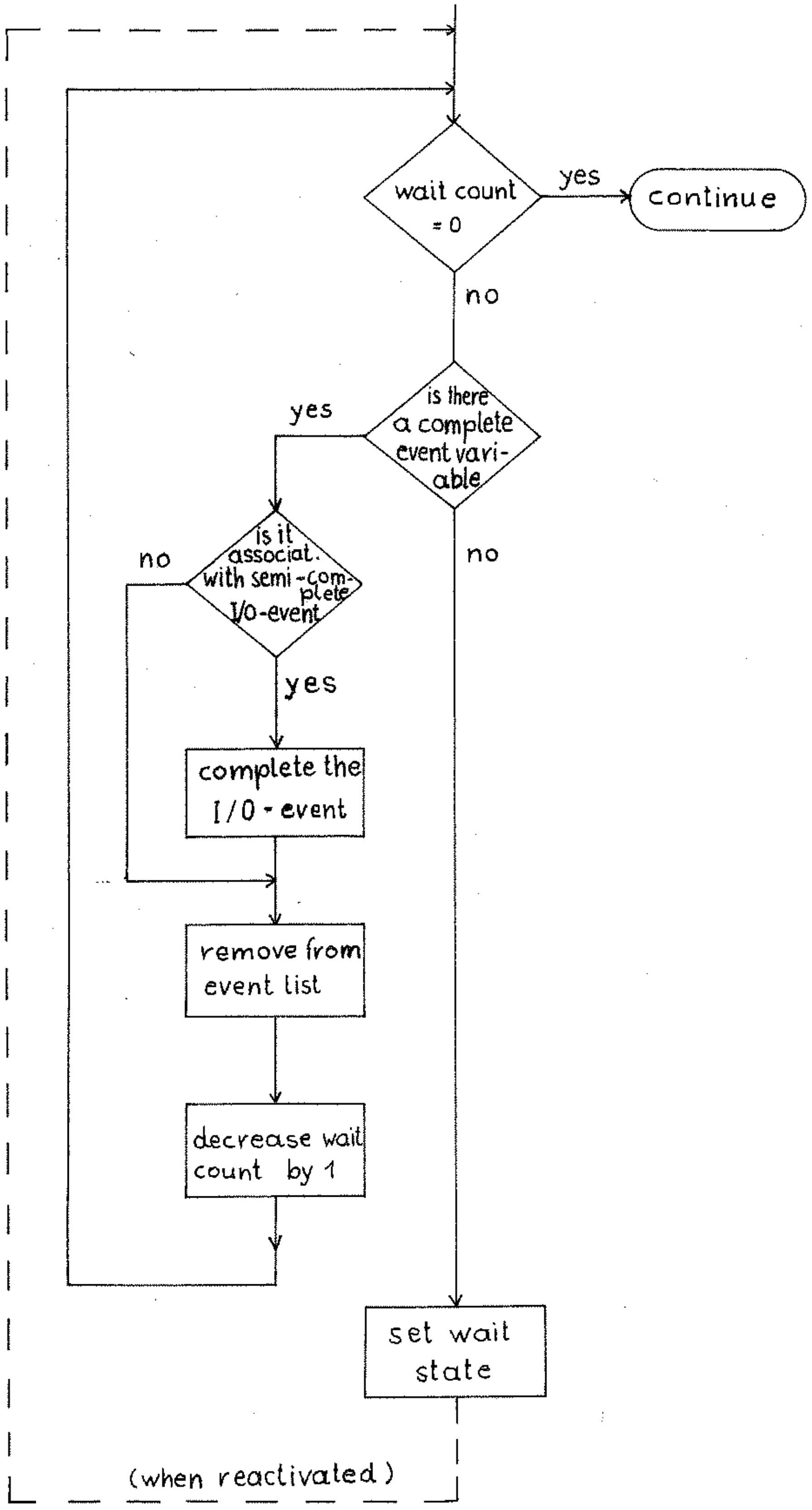


Fig. 7.7 The wait statement

When the task containing the wait statement is selected for execution by the priority scheduler it inspects whether at least one of the event variables to be waited for is recognized as complete. If so, that event variable is removed from the event list, the wait count is decreased by 1 and, if the event variable is associated with a semi-complete I/O-event, this I/O-event is fully completed. This is repeated until either the wait count is 0 or no more event variable in the event list is recognized as complete.

If the wait count is 0, the wait statement is said to be satisfied. It is finished and the task continues with the execution of the next statement. It should be noted that this may happen before the event list is exhausted (if the wait count was less then the number of event variables in the event list). In this case not all event variables in the event list need to be complete, in particular there may remain associated I/O-events which are not fully completed.

If the wait count is not yet 0, but there are no more event variables in the event list which can be recognized as complete, then the task containing the wait statement is set into the wait state, i.e., the wait state flag of its task-event specification $\underline{TE}$ is set (cf. 7.2). As a consequence, the priority scheduler will no more select this task for execution.

Whenever an action is performed by any task for which another task might be waiting (in particular the "complete" setting of any event variable and the semi-completion of any I/O-event) all tasks are reactivated, i.e., all wait state flags are removed. Then each formerly waiting task may again be selected by the priority scheduler for execution, it can inspect its event list whether there is now an event variable recognozed as completed or not. Depending on this it either continues the above described removing of event variables from the event list or it goes back into the wait state another time.

A similar mechanism applies also for all other situations in PL/I where a task has to wait for some specific action: The delay statement waiting for a specific time (this time is entered into a special component of the time and date part $\underline{TD}$ and all tasks are reactivated when it is exceeded by the time component of the state), the block epilogue waiting for the termination of all its daughter tasks, etc.

8. BLOCK ACTIVATIONS

Corresponding sections of /5/:

6.1 Block activation

6.2 Procedure call

3.2 Flow of control

The following abbreviations are used in this chapter:

AG	aggregate directory
arg	argument
ap	argument part
AT	attribute directory
b	unique aggregate name
BA,ba	block activation name
bpp	block prefix part
C, c	control
CI, ci	control information
CS, cs	condition status
CTL	controlled
D, d	dump
den	denotation
descr	descriptor
DN	denotation directory
EI, ei	epilogue information
elem	element
EV	attention environment directory
expr	expression
fct	function
FD	file directory

GEN, gen	generation
id	identifier
n	unique name
pref	prefix
ptr	pointer
ref	reference
ret-type	return type
sl	subscript list
st	statement

This chapter describes the dynamic handling of the block structure of PL/I within a single task. The block structure of a PL/I program (cf. 2.1) leads on interpretation to a dynamic system of nested block activations. At any time for each active task, there is one _block activation_ established in the state of the PL/I machine, called the _current_ block activation of that task. Whenever a task (including the main task at program start) or the interpretation of a block (begin block, procedure body, on-unit, attention) starts, a new block activation is established. When the interpretation of a block terminates the previous block activation is re-established (if there was any for the same task - the first block activation of a task terminates by terminating the task itself). All block activations which are established and not yet terminated are called _active_. The _dynamic descendants_ or _nested_ block activations of a given block activation are all those block activations which are established by actions of the given one and all those established by actions of these latter ones and so on. Within one task, at any time the systemof dynamic descendant block activations which are yet active is linear, i.e., each active block activation has at most one active immediate descendant in the same task. They may have more immediate descendants in daughter tasks (cf. 7.3,Fig. 7.5). It is a property of PL/I that no block activation is terminated until its dynamic descendants have been terminated.

8.1 THE DUMP D

Corresponding section of /5/:

3.2.1 The dump _D_

The current block activation is represented in the state of the PL/I machine by the following six local state components:

(1) the block activation name _BA_,

(2) the epilogue information _EI_,

(3) the condition status _CS_,

(4) the dump D,

(5) the control information CI,

(6) the control C.

These six state components contain all information which belongs to the current block activation and is obsolete on its termination. When a nested block activation is established, the local state components have to be saved for use after termination of the nested block activation, since the latter installs its own local state components.

The local state component of all active block activations which are not the current one are kept in the dump D. The dump is an object manipulated like a push-down stack, it maintains dynamically the history of the still active block activations. It consists of six components, namely the six local state components of the predecessor of the current block activation. Its dump component has again the same structure and consists of the local state components of the predecessor of that block activation, and so on. The dump of the first block activation is empty; the dump of the first block activation of a task (except the main task) is copied from the current dump of the mother task when the task is created. So the different levels of the dump represent the dynamic predecessors of the current block activation up to the first block activation of the program.

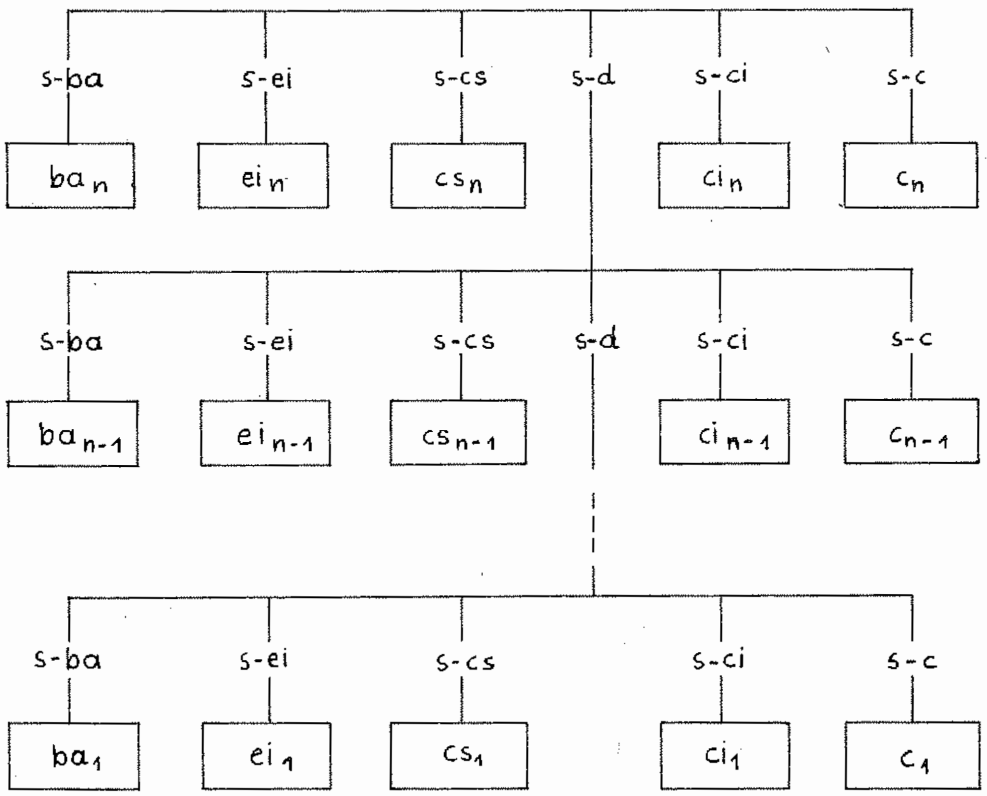


Fig. 8.1 The dump

When a new block activation is established the local state components of the previous block activation are copied as components into the dump. Thereby the former components of the dump automatically become components of the dump component of the dump, and so on; i.e., all parts of the dump are pushed down one level. Conversely, when a block activation is terminated, the components of the dump are copied into the local state components of the PL/I machine. All parts of the dump are thus popped up one level. This mechanism guarantees that all local state components are available as long as necessary, namely until the corresponding block activation is terminated, and that the right block activation is re-established when a block activation is terminated.

One should note that all information contained in the local state components (except the dump) is inherited into nested block activations, since they are copied into the dump and not destroyed on establishing a nested block activation. Afterwards, in general, the nested block activation will modify the inherited state components. Conversely, no information contained in the local state components (except the dump) is inherited back into outer block activations, since they are overwritten at block termination. The dump is left unchanged throughout a block activation, except in some cases of abnormal block termination, e.g., goto out of a block.

8.2 INTERPRETATION OF A BEGIN BLOCK

Corresponding sections of /5/:

6.1 Block activation

3.2.2 The block activation name BA

3.2.3 The epilogue information EI

As described in section 2.1.2, a begin block is a proper statement consisting of five components: a declaration part, a procedure body part, a condition prefix part, a statement list and a reorder option flag. Its interpretation consists of the creation of a new block activation, unique qualification of the locally declared identifiers in the text of the block, interpretation of the declarations and their installation in the state of the PL/I machine, interpretation of the statement list, and termination of the block activation.

In more detail, the following actions are performed, in the order given:

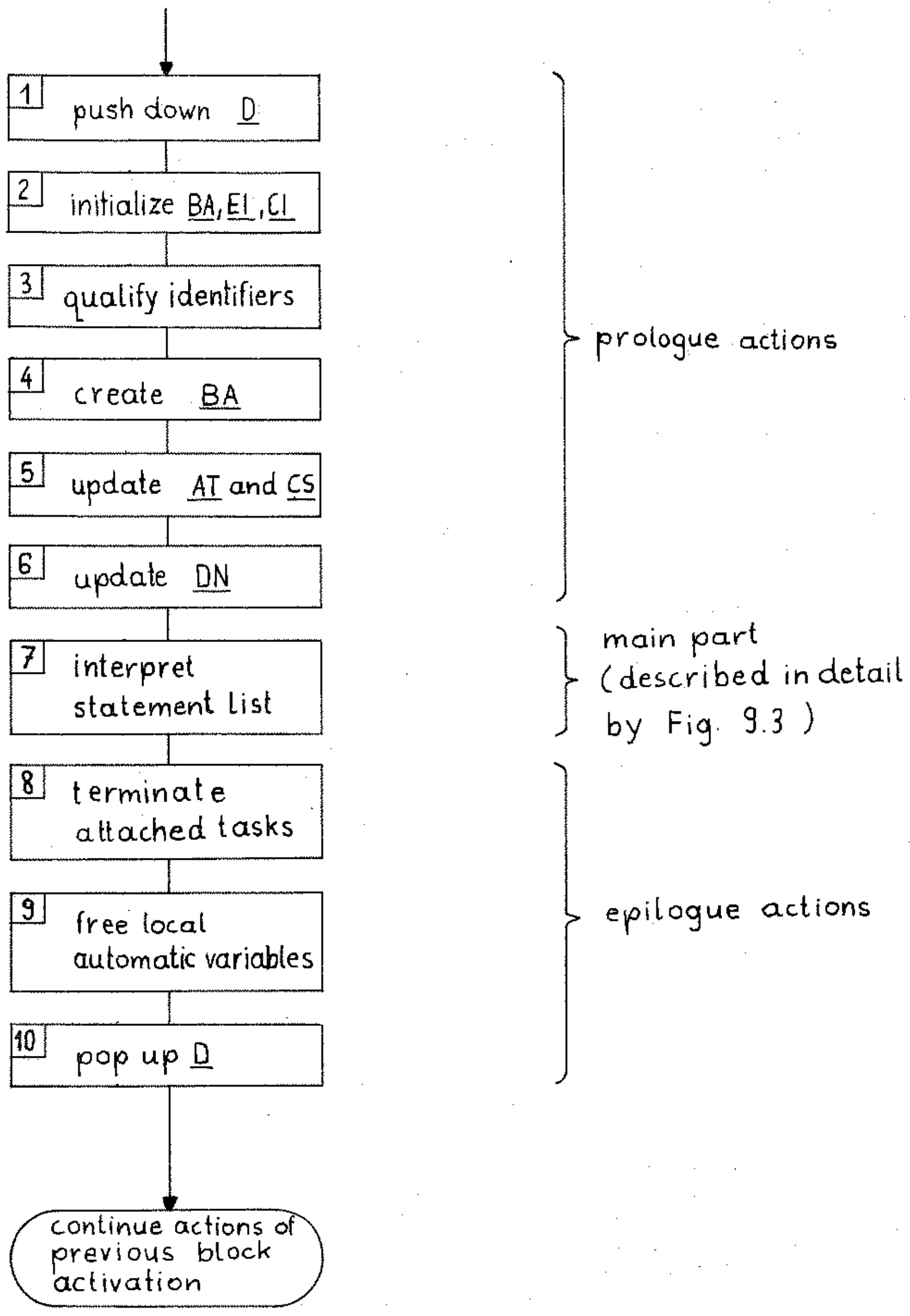

1 push down D
2 initialize BA, EI, CI
3 qualify identifiers
4 create BA
5 update AT and CS
6 update DN
7 interpret statement list
8 terminate attached tasks
9 free local automatic variables
10 pop up D
continue actions of previous block activation
prologue actions
main part (described in detail by Fig. 9.3)
epilogue actions

Fig. 8.2 Interpretation of a begin block

(1) The local state components of the previous block activation are copied into the dump as described in 8.1.

(2) The block activation name <u>BA</u> (cf. 8.2.2), the epilogue information <u>EI</u> (cf. 8.2.4) and the control information <u>CI</u> (cf. 9.4) are initialized for the new block activation.

(3) For each declaration contained in the declaration part of the block a unique name n is created and all occurrences of the declared identifier in the text of the begin block to be interpreted are qualified by this unique name n as described in 8.2.1.

(4) A unique block activation name <u>BA</u> is created (cf. 8.2.2).

(5) The attribute directory <u>AT</u> is updated by entering each declaration out of the declaration part of the block under its unique name n (cf. 5). The block prefix part of the condition status <u>CS</u> is updated by merging the previous one with the prefix condition part of the block (cf. 11.2.1).

(6) The denotation directory <u>DN</u> is updated by constructing and entering the denotation of each declaration out of the declaration part, under its unique name n, as described in 8.2.3.

(7) The statement list of the block is interpreted. This main part of the block interpretation is described in chapter 9.

(8) All tasks which are attached during the interpretation of the statement list and which are not yet completed are terminated abnormally. The information as to which tasks are to be terminated is found in the epilogue information <u>EI</u>. Interpretation is continued after termination of these tasks.

(9) The storage of all automatic variables allocated in this block activation is freed. The information as to which storage is to be freed is found in the epilogue information <u>EI</u>.

(10) The local state components of the previous block activation are copied back from the dump as described in section 8.1.

8.2.1 UNIQUE QUALIFICATION OF NAMES

The scope rules of PL/I require, that by a declaration the declared identifier receives its meaning for all occurrences within the text of the block containing the declaration. This meaning is inherited also into statically nested blocks as long as they do not contain another declaration for the identifier. If the same block is activated twice, both block activations are understood as completely independent of each other: All declarations give for both block activations different meanings to the declared identifiers. Another feature of PL/I is the fact that the interpretation of a piece of text may be postponed until another block activation than that one to which it belongs, i.e., that the meaning of identifiers occurring in this piece of text may differ from their meaning in the block activation established when the text is interpreted (e.g., declaration of a controlled variable, which is interpreted on allocation possibly in a nested block).

All these features are accomodated by the following simple mechanism: At the beginning of any block activation, a unique name n is created for each of its declarations. On the one hand, this unique name is used as selector to enter all information representing the meaning of the declared identifier into the different state components of the PL/I machine, in particular the attribute directory <u>AT</u> and the denotation directory <u>DN</u>. Thus the unique name n gives access to the meaning of the declared identifier id, associated with it. On the other hand, before any further use of the text of the block is made, each occurrence of the identifier id in this text (including all contained blocks) is qualified by adding the unique name n. Whenever, after this qualification has occurred, some part of that text is to be interpreted, not the identifier id itself, but the added unique name n is used to determine its meaning. When the activation of a nested block starts, those identifiers which are not redeclared keep the former unique names they had received earlier, while for the redeclared identifiers the unique names are overwritten by new ones according to the new meaning given by the redeclaration in the nested block. Any identifier occurring without matching declaration will receive no unique name by this mechanism and this will lead to an error when an attempt is made to interprete the text containing the occurrence. Any part of the text which is copied and interpreted during a later block activation retains its qualifying unique names and thereby the meaning of the contained identifiers. Thus the above mentioned scope rules of PL/I are obeyed fully by this mechanism.

<u>Example</u>:

```
DCL A (N) CTL, CHAR (3), N INIT (5);
BEGIN; DCL N CHAR (3) INIT ('ABC');
       ALLOCATE A INIT (N);
       ...
       END;
```

For the first declaration of N, in the outer block, a unique name n_1 is created and added to all occurrances of N. When the inner block is activated, for the second declaration of N a second unique name n_2 is created and added to all occurrences of N in the text of the inner block. Now, when the allocate statement is executed, the identifier N occurring in the dimension attribute is found to be qualified by n_1 and leads to a fixed point variable with value 5, while the identifier N occurring in the initial attribute is found to be qualified by n_2 and and leads to a character variable with value 'ABC'.

The described schema has to be slightly modified for the following reason: The most frequent occurrance of identifiers in the program text is in the context of references. In these cases not single identifiers, but identifier lists, so-called <u>qualified names</u>, occur possibly referring to components of structures. Not the main identifier, i.e., the head of the identifier list, but the whole identifier list determines to which declaration the reference refers.

Example:

```
DCL 1 S, 2 A, 2 B;
BEGIN 1 S, 2 A, 2 C;
      ...  S.A ...
      ...  S.B ...
      ...  S.C ...
      ...  S ...
      END;
```

The first declaration of S (in the outer block) gives rise to a unique name n_1, the second declaration of S (in the inner block) another unique name n_2. In the inner block the references S.A, S.C and S are to be qualified by n_2, referring to the second declaration, while the reference S.B is to be qualified by n_1 referring to the first declaration.

This is accomplished by qualifying all qualified names referring to the same declaration by adding the unique name of that declaration. A reference thus qualified has the following structure: besides the components described in 2.4.1 it has another component, namely the qualifying unique name n:

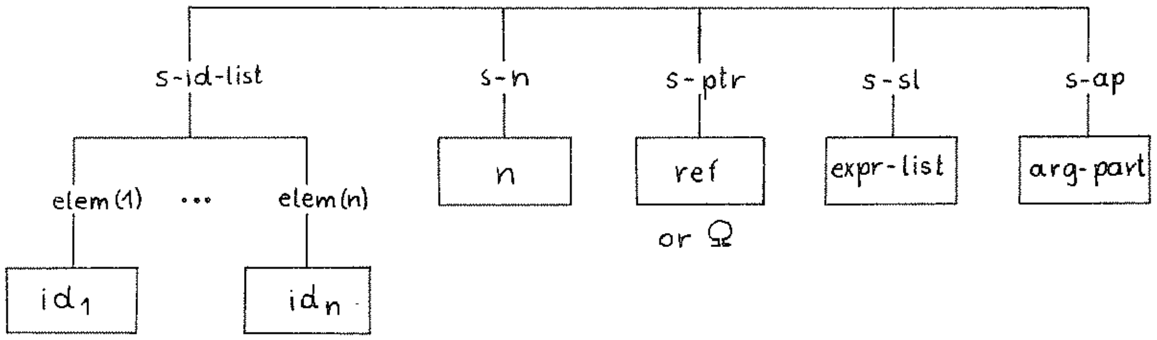


Fig. 8.3 Reference qualified by a unique name

8.2.2 THE BLOCK ACTIVATION NAME BA

For the meaning of some types of declarations (entry, label and format constants) the knowledge of their block activation, i.e., of the block activation containing the declaration, is essential. An entry or label constant may only be used by a call or goto statement in its own block activation or any of the dynamic descendants thereof. In the case of a goto statement the block activation of the label constant has to be re-established. A format constant may only be used by means of a remote format in its own block activation. Since it is possible by means of assignment to entry or label variables to transfer the values of entry, label and format constants into wrong block activations, one has to make the test when using these values.

For this purpose, each block activation which may contain declarations (i.e., the activation of a begin block or procedure body, but not of an on-unit or attention) is uniquely characterized by a unique name, the <u>block activation name BA</u>. This unique name is created and inserted into the state as local state component before the declarations are interpreted. It is then inserted into the denotation of entry, label and format constants. A later test for correct use of such a constant can then be performed by inspection of the current block activation name <u>BA</u> and the block activation names stacked at the different levels in the current dump <u>D</u>.

The block activation name of the outermost block activation of a task (including the main task) is left empty (i.e., Ω). This is tested by the goto statement to avoid a goto out of the task: A goto does not cross a level in the dump <u>D</u> whose block activation name is Ω. As a consequence, the block activation name component of the denotation of an external entry constant is always empty (since it is declared in the outermost block activation of the main task).

The block activation name of a block activation initiated by a condition or attention call is an asterisk (*). These block activations need not be uniquely identified since they do not contain declarations, nor must they be rejected as erroneous by a goto like the outermost block activations of tasks.

Special provisions have to be made for the case where an attention interrupt occurs in the period between the establishing of a block activation and the creation of its unique name <u>BA</u>, since it could happen in this case that a goto leads out of the block activation called by this interrupt. If the block activation name of the previous block activation would be intermediately taken over, such a goto might lead into the new block activation instead of the previous one; if the block activation name would be empty intermediately, a goto would run into error like a goto out of a task. Therefore, the block activation name of a new established block activation is intialized to * until a unique name has been created and inserted.

8.2.3 INTERPRETATION OF DECLARATIONS

The main action of the block prologue is to interpet the declarations of the block. That means to form the information about the meaning of the declared identifiers and to enter this information under the unique names associated with the declarations (cf. 8.2.1) into the different directories.

The texts of the declarations, as modified by the unique qualification of names (cf. 8.2.1), are entered into the attribute directory <u>AT</u> (cf. 5).

The donotations of the declarations are constructed and entered into the denotation directory <u>DN</u>. The denotations of the different types of declarations consist of all information which, besides the texts of the declarations themselves, constitutes the meaning of the declared identifiers. Its structure, for the single types of declarations, is described in chapter 5.

For a proper variable the denotation is a unique name b, the <u>aggregate name</u>, which is used to access the generation list of the variable in the aggregate directory <u>AG</u>. For static and controlled variables, the aggregate names were created by the prepass and entered into the text of the declarations. For automatic variables new aggregate names are created, and, additionally, storage is allocated and initialized.

For an internal entry constant the denotation is composed from the declared identifier itself, the corresponding procedure body to be found in the procedure body part of the block as described in 2.2.2, the current block activation name <u>BA</u> (cf. 8.2.2) and the block prefix part of the current condition status <u>CS</u>. For an external entry constant the same denotation was constructed at program start, entered into <u>DN</u> under a unique name n which was entered by the prepass into the text of all declarations of external entry constants of the same identifier throughout the text. Thus the block prologue has only to copy that denotation in the case of external entry constants.

For file constants and attentions the prepass created unique names as their denotations, serving as selectors to the file directory <u>FD</u> or the attention environment directory <u>EV</u>. These unique names were entered into the text of the declarations by the prepass.

For a label constant, the declaration is constructed from the current block activation name <u>BA</u> and the index list as given in the declaration. For a format constant, the denotation is constructed from the current block activation name <u>BA</u>, the format list and the identification which are both given in the text of the declaration, and the statement prefix part given in the declaration and merged with the current block prefix part.

For a defined variable, the denotation is produced by evaluating its aggregate attribute (cf. 2.2.1).

All other types of declarations (based variables, generic identifiers, builtin functions and programmer-named conditions) need no denotations. All information about their meaning is given by their declarations.

The interpretation of automatic and defined declarations requires expression evaluation. Since in these expressions locally declared identifiers may occur, provisions are made that no denotation of an automatic or defined variable is evaluated before the denotations of all those declarations have been entered into the denotation directory, which are needed for its evaluation. If this requirement is not satisfyable, since some declarations are mutually dependent (directly or indirectly) on each other in this sense, then the program is in error.

8.2.4 BLOCK EPILOGUE AND THE EPILOGUE INFORMATION <u>EI</u>

The <u>epilogue</u> of a block activation terminates all tasks attached during the block activation which are still active (cf. 7.4), frees the storage of all local automatic variables and re-establishes the previous block activation by copying the local state components from the dump as described in 8.1. The epilogue of a block activation is executed not only on normal termination of the block activation, i.e., on exhaustion of its statement list, but also on any occasion, when the block is to be terminated abnormally. These occasions are: the return statement (cf. 8.3.3), the goto statement (cf. 9.6) and the abnormal termination of the task (cf. 7.4).

There is a local state component, the epilogue information <u>EI</u>, which contains for each block activation all information needed by the epilogue for correct termination of the block activation. Additionally, it contains all information needed by a return statement to perform a correct return from a procedure body.

The epilogue information <u>EI</u> consists of the following components:

(1) The free set. The storage of all automatic variables and all dummy arguments is to be freed at the end of the block activation in which they are declared, or of the procedure to which they are passed, respectively. The free set of EI maintains the information as to which variables are to freed at block end. Initially it is the empty set for a begin block or the set of aggregate names of the dummy arguments for a procedure. Whenever an automatic variable is allocated, its aggregate name is added to the free set. On termination of a block activation all variables, whose aggregate names are contained in the free set, are freed.

(2) The task set. It has a similar purpose as the free set. Each task attached during a block activation is to be terminated abnormally at block end, if it has not been completed earlier. The task set is initially the empty set. Whenever a task is attached, its unique task name is entered into the task set of EI. At block end all tasks, whose unique task names are contained in the task set of EI and which are still active, are deleted.

(3) The block-activation-type. A return statement has to perform different actions depending on whether the current block activation is that of a begin block, a procedure body, or an on-unit. To recognize this distinction the epilogue information contains a component which is one of the elementary objects BLOCK, PROC, ON. This component is set when a block activation is established and never changed. For the block activation of an attention call, which never executes a return statement, this component is empty.

(4) The function denotation. If a procedure body is activated by a function reference, then, before the call, an aggregate name is created for a dummy variable for the function value to be returned. This aggregate name is reserved in the epilogue information of the called procedure to be used by a return statement for allocation of a dummy variable and assignment of the function value to this dummy. Since such a return statement may occur in nested begin blocks, the function generation is inherited into the epilogue information of nested block activations of begin blocks (but not of procedures). This component of EI is empty in procedures activated by call statements instead of function references (and in nested begin blocks).

(5) The return type. If a procedure body is activated by a function call, the return type of the called entry point is entered into the epilogue information to be used by a return statement for conversion of the function value and allocation of a dummy variable for it. The return type is inherited into the epilogue information of nested begin blocks in the same way as the function denotation. This component is empty in procedures activated by call statements instead of function references (and in nested begin blocks).

(6) The main procedure flag. If a return statement has to terminate the main procedure, i.e., the procedure activated by the initial call of the computation, the finish condition has to be raised first. The main procedure flag serves to indicate whether this is the case. It is set to * by the initial call and reset to Ω by all later procedure calls. Like the function denotation this flag is inherited into the nested block activations of begin blocks.

8.3 PROCEDURE CALL

Corresponding section of /5/:

6.2 Procedure call

A procedure call establishes a block activation of a procedure body by a call statement or function reference. This section describes all those actions performed during a procedure call which differ from the actions performed during a begin block activation as described in 8.2.

Both call statement and function reference consist essentially of a reference specifying the entry to be called and a list of expressions specifying the arguments to be passed to the parameter of the called entry.

The reference for the entry may refer to either an entry constant or a scalar entry variable or a function which returns an entry value. In all three cases an entry attribute is obtained out of the attribute directory AT: either the declaration of the entry constant, or the data attribute of the entry variable or the return type of the function. This entry attribute consists essentially of two components (further components for the three different cases are of no interest here): A parameter descriptor list and a return type.

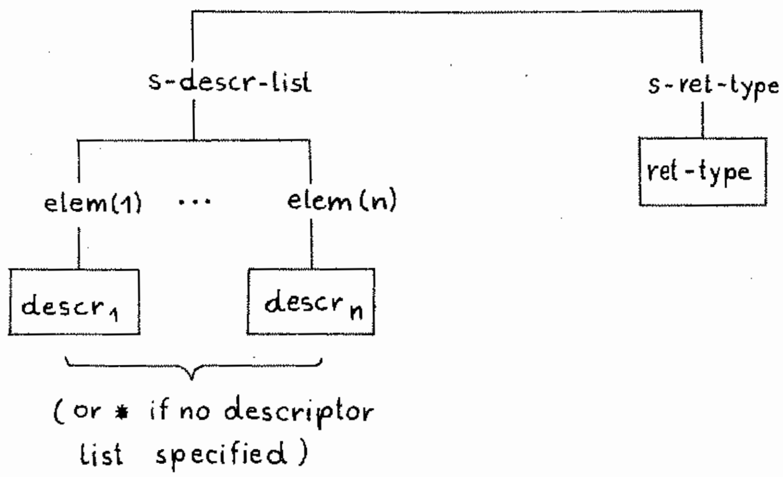


Fig. 8.4 Entry attribute

The evaluation of the entry reference yields as value, in all cases, the unique name of an entry constant: either that of the referenced entry constant itself, or that of an entry constant which was assigned to the referenced entry variable, or that of an entry constant returned by a function call. Applying this unique name to the denotation directory DN one obtains an entry denotation, namely the denotation of the entry constant to be called. The entry denotation consists of:

(1) the procedure body to be called,

(2) an identifier specifying the called entry point into the procedure body,

(3) a block activation name (to be used only for testing as described in 8.2.2),

(4) a block prefix part to be inherited into the condition status of the called procedure.

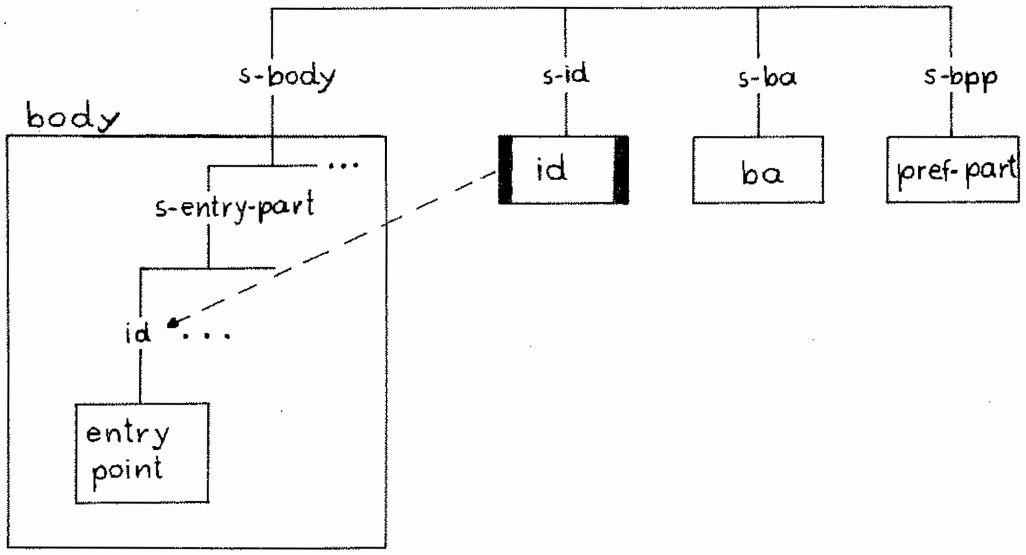


Fig. 8.5 Entry denotation

The entry attribute and entry denotation obtained in this way, together with the list of argument expressions, contain all information necessary for performing the procedure call. The entry attribute, which is obtained for the reference out of the attribute directory without any expression evaluation, is used only in the calling block activation. The entry denotation, which is obtained out of the denotation directory after evaluation of the reference, is only used in the called block activation.

A procedure call differs from a begin block activation in the following main points:

(1) Instead of a begin block occurring directly as the statement to be interpreted, a procedure body contained in the entry denotation has to be activated.

(2) The expressions given in the call statement or function reference are passed as arguments to parameters specified in the procedure body, as described in detail in 8.3.1.

(3) In case of a function reference a value is returned after termination of the block activation as described in 8.3.2.

(4) The interpretation of the statement list is not necessarily started at the beginning, but at a point somewhere within the

statement list. This point is determined by the statement location component of the entry point, which is specified by the identifier of the entry denotation (cf. 2.1.3 and Fig. 8.5). The start of the interpretation of the statement list, using an index list determining the statement location, is performed by the mechanism of the goto statement (cf. 9.6).

8.3.1 ARGUMENT PASSING

For passing the arguments of a procedure call to the parameters of the called procedure body the following information is available:

(1) the _argument expressions_ in the call statement or function reference,

(2) the _parameter descriptors_ in the entry attribute (if * is specified instead of a descriptor list, all descriptors are assumed to be *),

(3) the _parameter identifiers_ in the specified entry point of the procedure body,

(4) the _parameter declarations_ in the declaration part of the procedure body by means of the parameter identifiers.

For each single argument the following actions are performed:

(1) Before the call, i.e., in the calling block activation, the decision is made as to which of the following three types of action is performed. This decision is based on the syntactical form and attributes of the argument on the one hand and the parameter descriptor on the other hand. It should be noted that this decision is made without any expression evaluation (i.e., can be performed "at compile time" in an implementation).

(a) _Passing of denotation_ (controlled stack). This is performed if: the argument expression in fact is a reference referring to the complete declaration (and not any sub-component) of a controlled variable; the parameter descriptor specifies controlled storage class (or is *); and the aggregate attributes of the argument and of the parameter descriptor (if it is not *) match, except for extents.

(b) _Passing of generation_. This is performed if: the argument expression in fact is a reference to a variable (or to a sub-component thereof); the parameter descriptor does not specify controlled storage class; the aggregate attributes of the argument and of the parameter descriptor (if it is not *) match; and, provided the parameter descriptor is specified as connected, the argument refers to connected storage.

(c) _Passing of value_ (dummy variable). This is performed in all other cases, if the parameter descriptor does not specify controlled storage class.

(2) Before the call, the argument is evaluated resulting in an object consisting of an aggregate name b and a type designator (one of the three elementary objects CTL, GEN, DUMMY denoting the three types of passing).

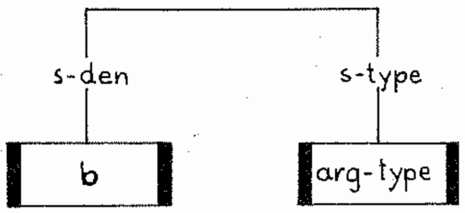


Fig. 8.6 Evaluated argument to be passed

The evaluation uses the argument expression and the parameter descriptor. It depends on the type of passing:

(a) Passing of denotation. In this case the aggregate name of the controlled variable referred to by the argument expression is taken as aggregate name of the argument.

(b) Passing of generation. In this case the generation referenced by the argument expression is evaluated, entered into the aggregate directory under a newly created unique name, and this unique name is taken as aggregate name of the argument.

(c) Passing of value. In this case, first the argument expression is evaluated and, if a parameter descriptor is present, converted to the type of the parameter descriptor. If the parameter descriptor or the argument expression is an aggregate, the evaluation and conversion is performed by expansion into the scalar components (cf. 10.2.8), using the parameter descriptor (if present) or the aggregate attribute of the expression as master aggregate. The resulting operands are intermediately set aside in an auxiliary object, called dummy operand. Second, the evaluated aggregate attribute of the resulting aggregate is determined from the dummy operand. It may happen that for an array of strings each single component in the dummy operand has a different string length. In this case the maximum length is taken. Third, a dummy variable with the determined evaluated aggregate attribute is allocated and its generation is entered into the aggregate directory AG under a newly created unique name, which then becomes the aggregate name of the argument. Fourth, the single scalar components of the evaluated dummy operand are assigned to the corresponding scalar components of the dummy variable.

Example:
```
DCL Q ENTRY((3) CHAR(*)),
    I INIT(0);
P:PROC RETURNS(CHAR(*));
  I=I+1;
  RETURN(SUBSTR('ABCDE',1,I));
  END P;
CALL Q(P);
```

For the call of Q the argument expression P has to be evaluated using the parameter descriptor (3) CHAR(*). In the first step the argument expression P is expanded into three scalar components and evaluated, i.e., the procedure P is invoked three times returning the three scalar values 'A', 'AB', 'ABC'. These three values (more exactly: the corresponding operands) constitute an intermediate dummy operand. In the second step, the evaluated aggregate attribute (3)CHAR(3) is determined from this, i.e., an array with bounds 1:3 and scalar components being character strings of length 3. In the third step, a dummy variable for this aggregate is allocated and, in the fourth step, the components of the dummy operand are assigned to it, yielding the values 'A ', 'AB ', 'ABC'.

(3) After the call, i.e., in the block activation established by the procedure call, the parameter identifier, like all locally declared identifiers, is associated with a unique name n (cf. 8.2.1) and the attributes of the parameter declaration are entered into <u>AT</u>. The argument, which has been evaluated before the call as described, is passed to the called block activation. Its aggregate name is, after testing of the generation of the argument against the parameter declaration, entered into the denotation directory <u>DN</u> as the denotation of the parameter. Thus, in the called block activation the parameter has the attributes of the parameter declaration and the denotation resulting from the above described argument evaluation.

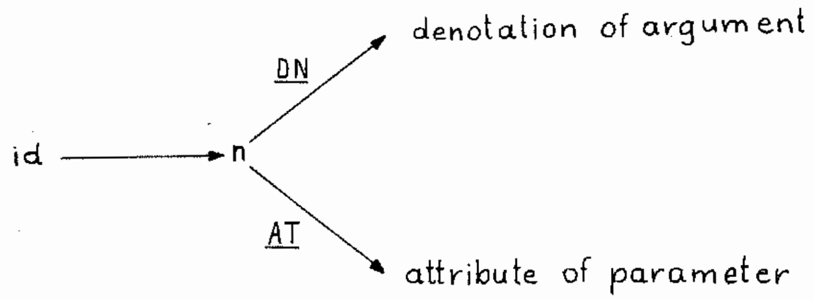


Fig. 8.7 Connection of parameter identifier with denotation and attribute

There is one exception from this general rule: For controlled arguments passed to non-controlled parameters, after the call, the last generation instead of the complete generation list in <u>AG</u> is connected with a newly created unique aggregate name b, which becomes the denotation of the parameter, i.e., the passing of a generation (instead of a denotation) is simulated.

Example

The effect of the three different types of argument passing may be illustrated by the following example:

```
DCL A CTL FIXED INIT(0),
    P ENTRY(CTL FIXED, FIXED, FIXED);
ALLOCATE A;
CALL P(A, A, (A));
P:PROC(X, Y, Z);
  DCL X CTL FIXED, Y FIXED, Z FIXED;
...
END P;
```

To all three parameters X, Y, Z the same argument expression (or nearly the same) corresponds, but to X the denotation is passed, to Y the current generation and to Z the current value 0. After argument passing the chains for the four identifiers A, X, Y, Z from the identifier via unique name, denotation directory and storage to the value 0 are as illustrated in the following figure:

$$\begin{tikzcd}
A \arrow[r] & n \arrow[r, "\underline{DN}"] & b \arrow[r, "\underline{AG}"] & \langle gen, \dots \rangle \arrow[ddr, "S"] & \\
X \arrow[r] & n_1 \arrow[ur] & & & \\
Y \arrow[r] & n_2 \arrow[r] & b_2 \arrow[uur] & & 0 \\
Z \arrow[r] & n_3 \arrow[r] & b_3 \arrow[r] & \langle gen, * \rangle \arrow[ur] &
\end{tikzcd}$$

Fig. 8.8 The three types of argument passing

Obviously X shares the denotation, Y the generation and Z the value with A. Any assignment to A in the called procedure will change the common current value of A, X, Y (as long as no allocation or freeing of A has occurred), but not the value of Z. Any allocation or freeing of A will change the common current generation of A and X, but not the generations of Y and Z. A and X differ only with respect to their attributes in <u>AT</u>.

8.3.2 FUNCTION REFERENCE

The interpretation of a function reference, occurring during expression evaluation, differs from the interpretation of a call statement only in the fact that provisions are made to return an operand for the function value.

For this purpose there are two return types available: one (the <u>outer return type</u>) in the entry attribute to be used in the calling block activation, and another (the <u>inner return type</u>) in the entry point in the procedure body to be used in the called block activation. If both return types do not match (they have to be identical except for string length if specified by *), the program is erroneous.

The returning of the operand by the function reference is performed in the following way:

(1) Before the call, a unique name b, to be used as aggregate name of a dummy variable, is created and passed to the called procedure. This unique name, and the outer return type, is the only relevant information known to the calling block activation.

(2) When establishing the called block activation the aggregate name b and the inner return type are entered into the epilogue information EI of the called block activation. They are also inherited into nested block activations of begin blocks (cf. 8.2.4).

(3) A return statement in the called block activation (or in a nested one) evaluates its return expression and converts the result operand to the inner return type found in the epilogue information EI. Then it allocates a dummy variable, taking the evaluated aggregate attribute from the evaluated operand, and enters its generation into the aggregate directory AG under the aggregate name b found in the epilogue information EI. Finally it assigns the evaluated operand to this dummy variable.

(4) After return from the called block activation, the calling block activation accesses the dummy variable by means of the aggregate name b; it takes the operand, tests its aggregate attribute against the outer return type and frees the dummy variable.

8.3.3 RETURN FROM A PROCEDURE

A procedure called by a call statement may be terminated regularly either by a return statement without expression specified or by coming to the end of the statement list of the procedure body. A procedure called by a function reference may be terminated regularly only be a return statement with an expression specified. Irregularly it may be terminated by a goto statement or any kind of abnormal task termination.

To have only one case of regular termination, the end of the statement list of a procedure body is handled as a return statement without a specified expression.

A return statement may occur within nested begin block activations; in this case it also has to terminate all begin block activations nested in the innermost procedure activation.

If the procedure to be terminated is the main procedure of the program, the finish condition has to be raised before any block activation is terminated.

To ensure the availability of all necessary information, the epilogue information EI contains the function denotation (or Ω in the case of a procedure called by a call statement) the return type (or Ω in the case of a procedure called by a call statement) and the main procedure flag, which are inherited into nested begin block activations, and the block activation type, which is not inherited (cf. 8.2.4).

Using this information, a return statement works as described in Fig. 8.9.

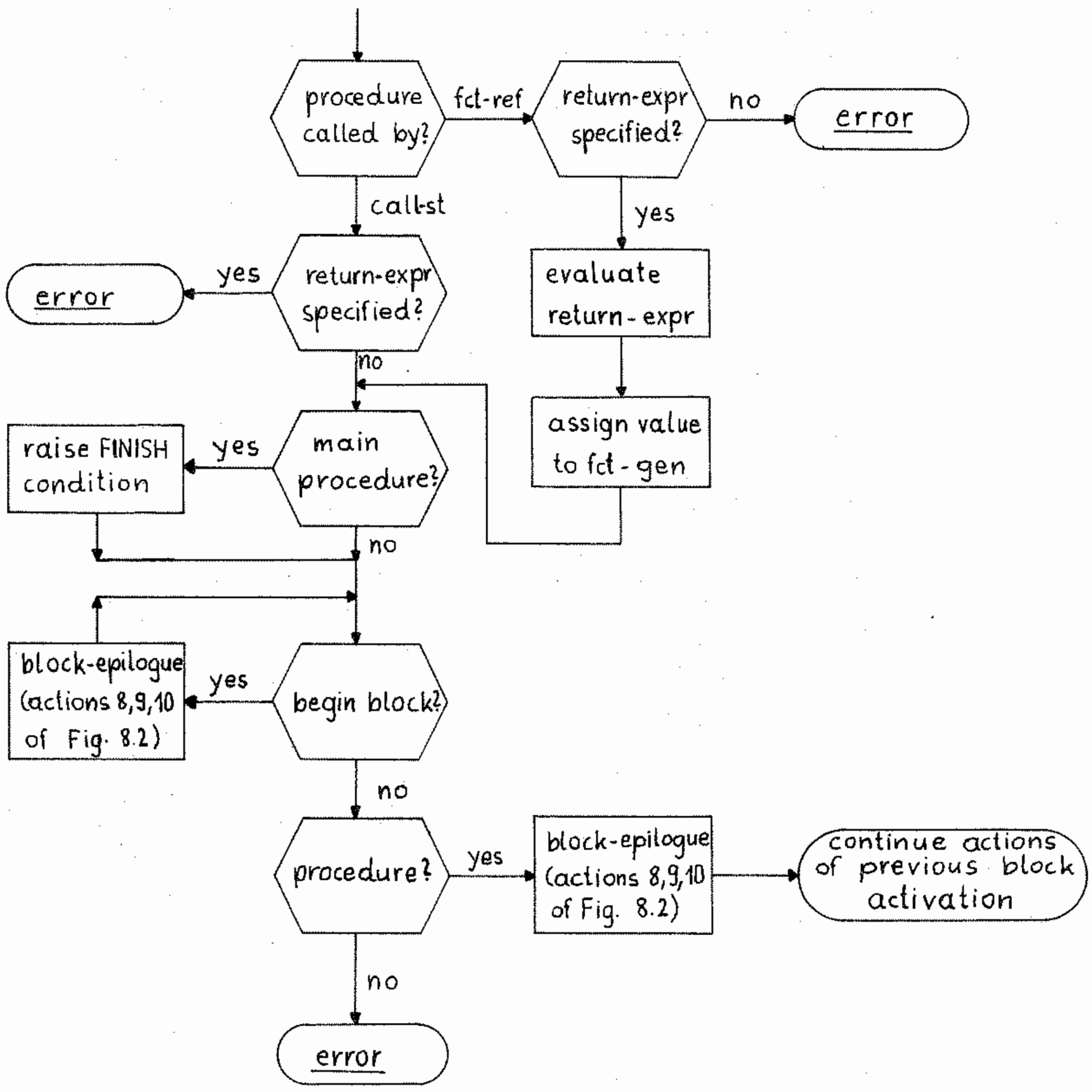


Fig. 8.9 Return statement

8.3.4 GENERIC SELECTION

It is possible in PL/I to declare a generic identifier for a family of entry references and to use the generic identifier in call statements and function references instead of an entry reference. In such a case, before the call is performed the generic selection takes place which selects, governed by the attributes of the arguments of the call, one out of the entry references of the family. The selected entry reference is to be evaluated and the resulting entry constant to be called as described in the previous section.

The declaration of a generic identifier is the list of the generic members, each consisting of an entry reference and a list of possibly incomplete parameter descriptors. The generic selection has to select the entry reference of the first generic member in the list, whose parameter descriptors (as far as specified) are identical with the attributes of the arguments of the call.

More specifically, this is performed in three steps:

(1) The attributes of the results of the argument expressions are determined (without evaluating the expressions themselves).

(2) If the attribute of any argument (or a scalar component thereof) turned out to be of type entry, the problem arises whether the generic selection is to be made under the assumption that the entry itself is to be passed as argument, or that it shall be invoked and the result passed. For this decision a prescan is made: For each attribute of an argument (or scalar component of an argument attribute) of type entry the generic members are inspected whether any of them specifies the type entry (or no type at all) at the corresponding position in the descriptor list. If so, the selection is done under the assumption that the entry itself is to be passed. If not, it is assumed that the entry is invoked (with empty argument list) repeatedly until the result is not of type entry anymore. For the generic selection this means that the entry in the list of argument attributes is to be replaced by its final (non-entry) return type, before the list of argument attributes is used as basis for generic selection.

Example:

```
DCL G GENERIC (P1 WHEN(1,2 FIXED,2 FIXED),
               P2 WHEN(1,2 ENTRY,2 FIXED)),
   1 S, 2 E1 ENTRY RETURNS(FIXED),
        2 E2 ENTRY RETURNS(FIXED);
CALL G(S);
```

Before the generic selection, the prescan inspects for each scalar component of the argument S whether there is a corresponding position in the descriptor list of any generic member of the type entry or not. Based on this inspection the first component is assumed as entry type, while for the second the return type is taken. I.e., the generic selection is performed with the modified argument attribute

```
1, 2 ENTRY, 2 FIXED,
```

which then selects the member P2.

(3) With the so modified argument attribute list the proper generic selection is performed: One generic member after the other is tested (in the given order) whether the number and attributes of its parameter descriptors, as far as specified, are the same as those of the arguments. The entry reference of the first generic member satisfying this condition is selected and replaces the generic identifier.

9. FLOW OF CONTROL WITHIN A SINGLE BLOCK ACTIVATION

Corresponding sections of /5/:

6.3 Sequential interpretation of statements

3.2.4 The control information <u>CI</u>

6.5 Groups

6.4 The goto statement

The following abbreviations are used in this chapter:

ba	block activation name
<u>C</u>,c	control
<u>CI</u>,ci	control information
<u>CS</u>	condition status
<u>D</u>	dump
<u>DN</u>	denotation directory
elem	element
F	false
i	integer
spp	statement prefix part
st	statement
T	true

As described in 2.3 the statement list of a block constitutes a rather complicated system of nested statements, since some types of statements may themselves contain statements of any type or even lists of statements of any type. The present chapter describes the flow of control of the PL/I machine through this system of statements.

The interpretation of a begin block or of an on-unit is performed by establishing a new block activation and, after its termination, re-establishing the old one and continuing, as described in chapter 8. Thus, the flow of control within a single block activation remains to be described. The normal flow is influenced by:

(1) the sequencing of statements within a statement list,

(2) the nesting of statement lists within statements,

(3) the nesting of statements occurring as then and else alternatives in if- and access statements (the else alternative in an access statement is on the whole treated in the same way as that in an if-statement; it is omitted in this chapter),

(4) the iteration specifications of groups.

The flow of control is governed by a local state component, the <u>control information</u> <u>CI</u>. It reflects the current status of the PL/I machine with respect to these four points.

Additionally, the flow of control may be modified abnormally by means of the goto statement. This is performed essentially by modifying the control information <u>CI</u>.

9.1 SEQUENTIAL EXECUTION OF STATEMENTS

Corresponding sections of /5/:

6.3.1 Statement list

6.3.3 Interpretation of a single statement

The sequential interpretation of statements within a statement list is governed by two components of the control information <u>CI</u>, the <u>text</u> and the <u>index</u>. Whenever a statement out of a statement list (but not a nested statement contained within it) is under execution the text is that statement list and the index is the number of the currently executed statement within the list; e.g., when the third statement of a statement list is executed the text is the statement list and the index is the integer 3.

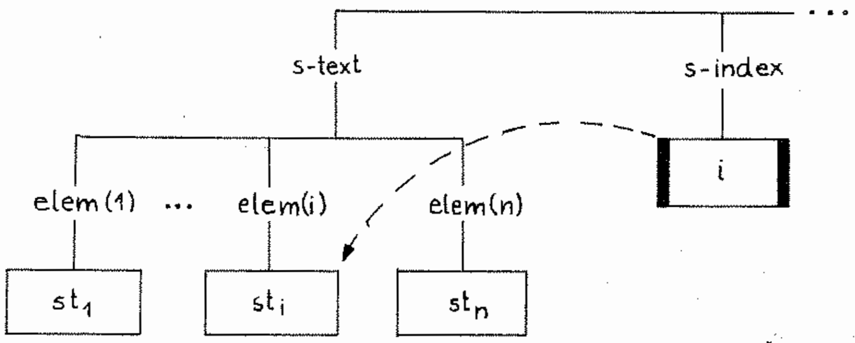


Fig. 9.1 Text and index components of <u>CI</u> on execution of a statement list

During the execution of a statement list, the text component of <u>CI</u> is in general left unchanged, while the index is always updated between two statement executions.

Whenever the execution of a statement (except the last one of the statement list) has been terminated, the index is increased by 1, the statement denoted by the new index is taken from the text and executed.

The execution of a single statement consists of:

(1) updating of the statement prefix part of the condition status CS, merging the block prefix part of CS and the condition part of the statement (cf. 11.2.1),

(2) raising of the check condition for the labels of the statement (cf. 11.5),

(3) execution of the proper statement, which depending on the statement type is described in the individual chapters of this document.

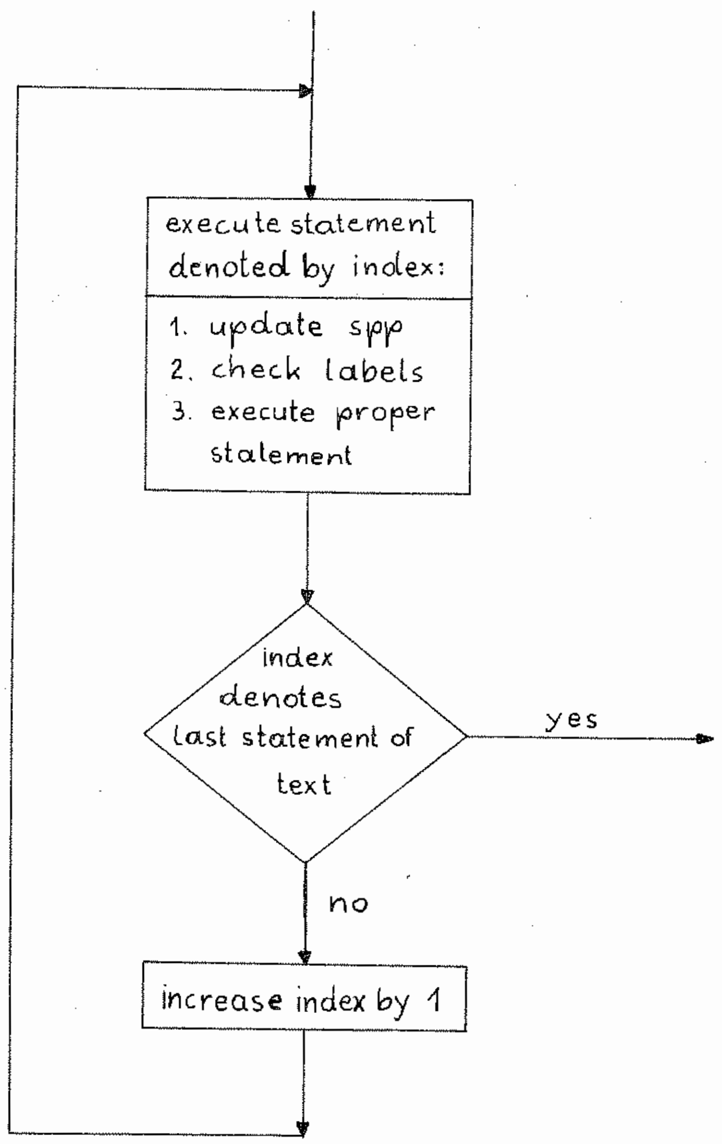

execute statement denoted by index:
1. update spp
2. check labels
3. execute proper statement
index denotes last statement of text
yes
no
increase index by 1

Fig. 9.2 Sequential execution of statements, governed by text and index components of CI

9.2 NESTING OF STATEMENT LISTS

Corresponding section of /5/:

6.3.1 Statement list

When a statement list is to be executed it is entered into the text component of CI, the index component of CI is initialized to 1 and the mechanism described in 9.1.1 is started.

These actions are not sufficient in the case where a statement list is to be executed during the execution of a statement which itself is a member of a containing statement list. For in this case the text and index components of CI keep the information needed for the sequential execution of the statements of the containing statement list. This information would be lost by overwriting, if no special provisions were made when the nested statement list is executed. In order to keep the text and index for the containing statement list and also information in the control specifying how to continue after termination of the nested statement list, the control information CI is handled as a stack (similarly to the dump D, cf. 8.1): Whenever the execution of a statement list starts, before the text and index components of CI are overwritten, the complete current control information CI and control C are copied into two additional components of CI. When the last statement of the nested statement list has been executed, these two components are reinstalled as state components CI and C, and the execution of the containing statement list continues correctly.

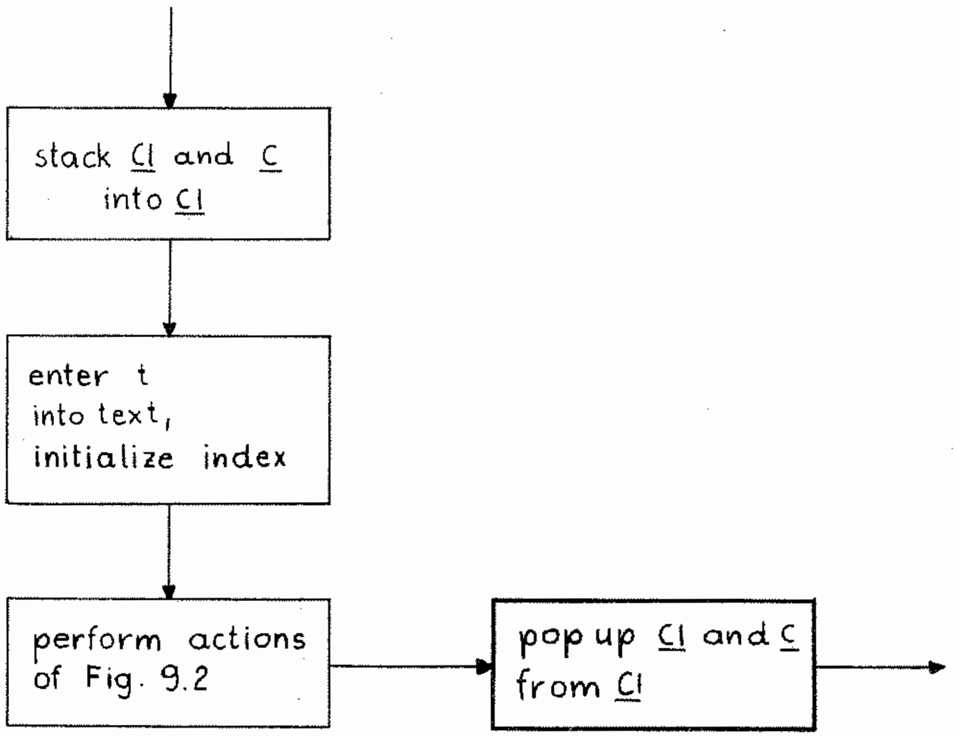

stack CI and C into CI
enter t into text, initialize index
perform actions of Fig. 9.2
pop up CI and C from CI

Fig. 9.3 Interpretation of a statement list t

Thus, the control information (apart from one special component, which is used only in edit directed stream I/O-statements) consists of four components: The current text and index, and the control information and control of the containing statement list. Again this control information of the containing statement list consists of four such components, and so forth. Each level in the control information represents one level in the system of nested statement lists.

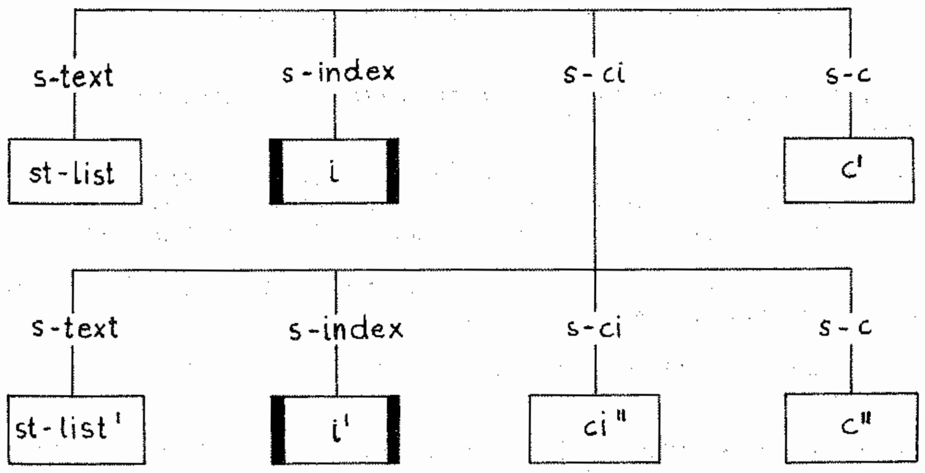


Fig. 9.4 Control information for nested statement lists

9.3 THE IF-STATEMENT

Corresponding section of /5/:

6.3.2 The if-statement

The if-statement (and access statement) introduces into the language a form of statement nesting differing from the nesting of statement lists. In order to reflect this form of statement nesting also, the concepts of text and index components of CI is slightly modified: The text may not only be a statement list but also an if-statement. In this case, the index is not an integer, but a truth value, the index T denoting the s-then component of the text and the index F denoting the s-else component.

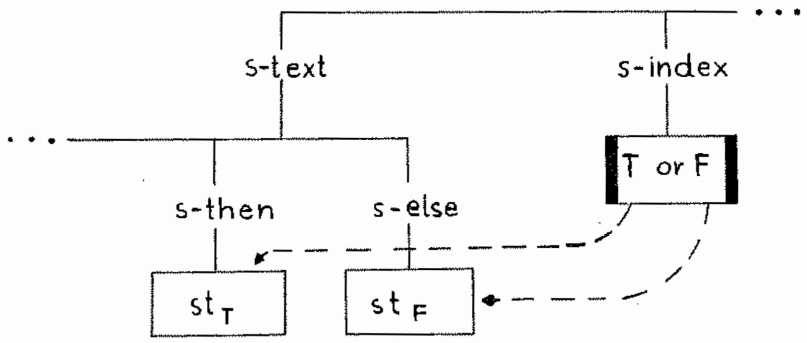


Fig. 9.5 Text and index components of CI on execution of an if-statement

The execution of an if-statement t causes the following actions to be performed:

(1) The decision expression of the if-statement is evaluated and converted to a truth value truth.

(2) The same actions, i.e., pushing down the control information for one level, are performed as described for the execution of a statement list in 9.1 and 9.2 with the following changes:

(a) the if-statement t (instead of a statement list) is entered into the text component of CI;

(b) the index is initialized to the truth value truth (instead of the integer 1);

(c) the meaning of "index denotes a statement out of text" is extended as explained above;

(d) both the s-then and s-else components are considered as "last" statements of text (Fig. 9.2), i.e., the control information CI is popped up after termination of either of them.

9.4 STRUCTURE OF THE CONTROL INFORMATION CI

Corresponding section of /5/:

3.2.4 The control information CI

As described in the previous sections, the control information CI consists of four components: text, index, control information and control, where the contained control information again consists of these four components, and so forth. Each contained level in the control information represents a containing level in the system of nested statements. It ends up with the level representing the outermost statement list of the current block activation; this one did not stack a control information, but the control causing finally the block epilogue.

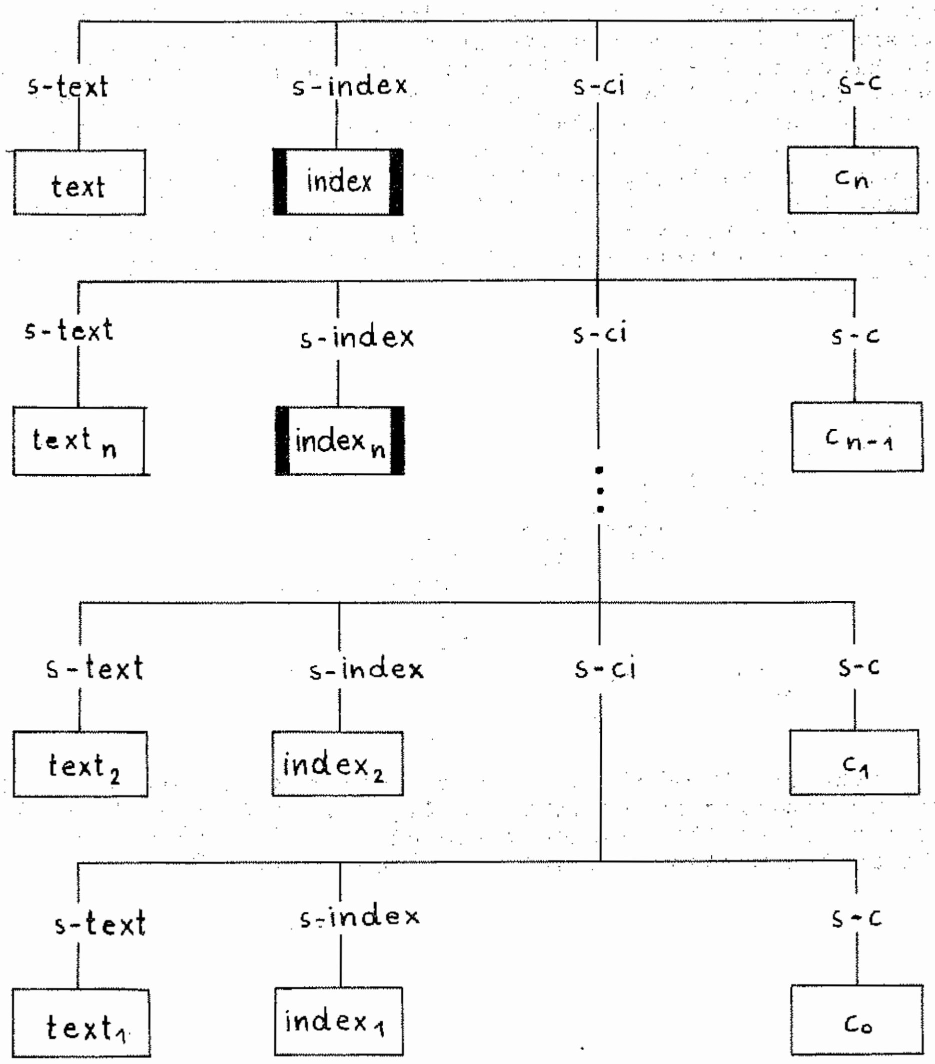


Fig. 9.6 Structure of the control information

The structure of the control information is illustated by Fig. 9.6. $text_1$ is the outermost statement list of the current block activation, $index_1$ is an integer denoting the statement st_1 of $text_1$ currently under execution, and c_0 is the control specifying the actions of the block prologue to be performed after termination of the execution of $text_1$. For each integer i between 1 and n, either $text_i$ is a statement list and $index_i$ an integer, or $text_i$ is an if-statement and $index_i$ a truth value. $index_i$ denotes that statement st_i out of $text_i$ which is currently under execution. The proper statement of st_i is either a statement list or a group containing a statement list or an if-statement. This statement list or if-statement is the component $text_{i+1}$. The control c_{i-1} specifies the actions to be performed after termination of the execution of $text_i$.

In this way the control information CI denotes exactly the innermost statement currently being executed within the system of nested statements of the current block activation. Since the component $text_{i+1}$ is always already uniquely determined by $text_i$ and $index_i$, all except the outermost ($text_1$) text components are redundant. They are always copied for convenience of use. In fact, the innermost currently executed statement is determined uniquely by $text_1$ and the list of indices: $<index_1, index_2, \ldots, index_n, index>$. This way of localizing a statement relative to the statement list of a block by a list of indices is used in the declaration of label constants (cf. 2.2) and in the execution of the goto statement (cf. 9.6).

Example:

```
BEGIN;L1 : ... ;
      L2 : ... ;
      L3 : DO; L31 : ... ;
               L32 : IF ...
                     THEN L32T : ... ;
                     ELSE L32F : DO I = 1 TO N;
                                 L32F1 : ...;
                                 L32F2 : ...
                                 END;
               L33 : ...;
           END;
END;
```

If the statement labeled L32F1 is currently under execution the control information is as given in Fig. 9.7 (writing always the labels instead of the labeled statements). The index list localizing the statement under execution is <3,2,F,1>.

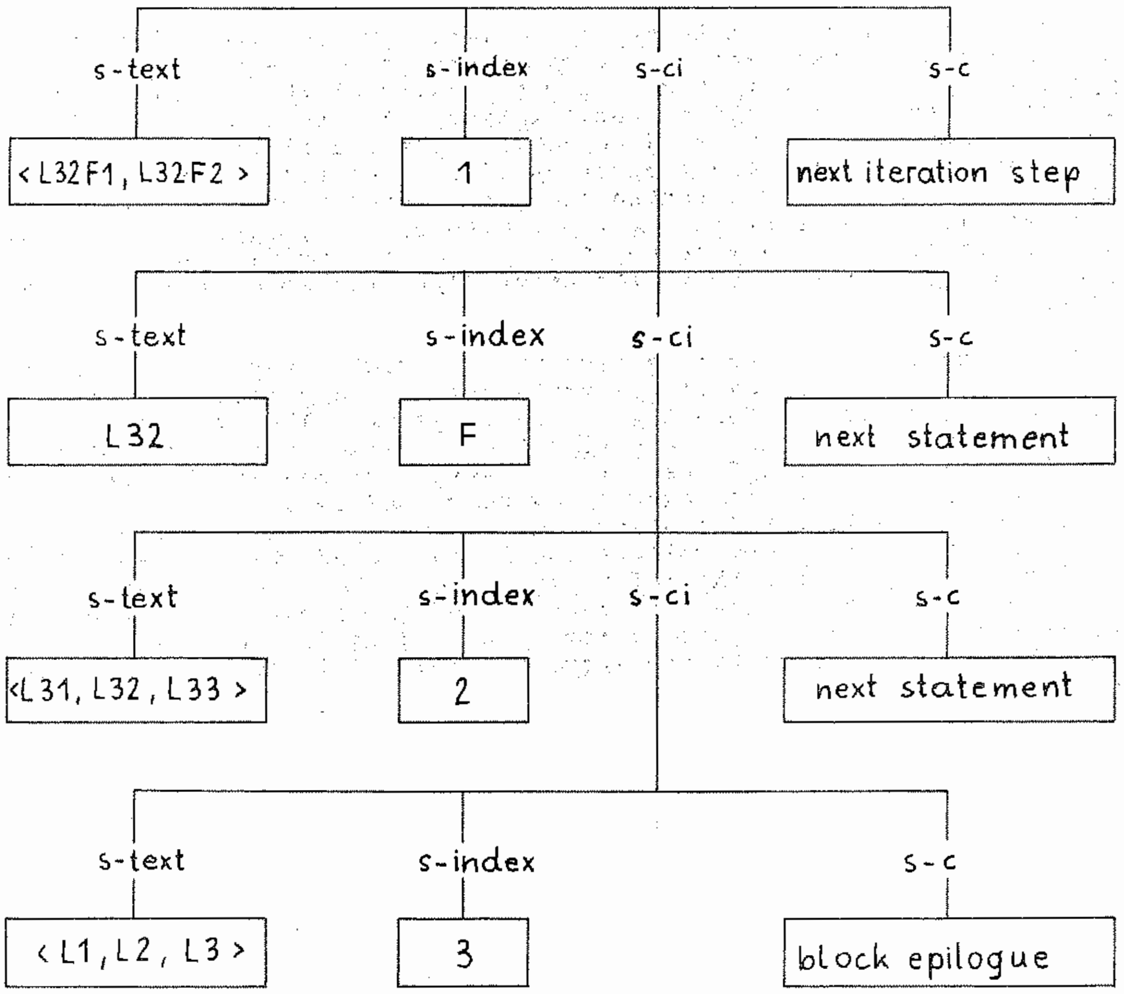


Fig. 9.7 Example of control information CI

9.5 GROUPS

Corresponding section of /5/:

6.5 Groups

A group is a proper statement specifying repeated execution of a statement list. There are two types of groups, the while group and the controlled group. The text of a group consits of the statement list to be iterated and an iteration specification. In the case of a while group the statement list is executed repeatedly until a given condition is satisfied, i.e., until the evaluation of a given expression yields "true". In the case of a controlled group after each execution of the statement list the value of a given variable, the controlling variable, is incremented by a given value, the statement list is executed repeatedly until the value of the controlling variable exceeds a given value.

The execution of a group is performed in such a way that all actions controlling the iteration of the statement list are performed at the level of the control information CI which is installed when the execution of the group starts. Each time when the iterated statement list is to be executed, the control information is stacked for one level, i.e., the statement list is executed exactly as described in 9.2. During the execution of the iterated statement list, the control component of CI specifies the actions controlling the iteration of the statement list, in particular, it contains the information about the current status of the iteration. Each time when the exectuion of the iterated statement list terminates, the control information is popped up for one level as described in 9.2. Thereby the iteration control is performed at the popped up level.

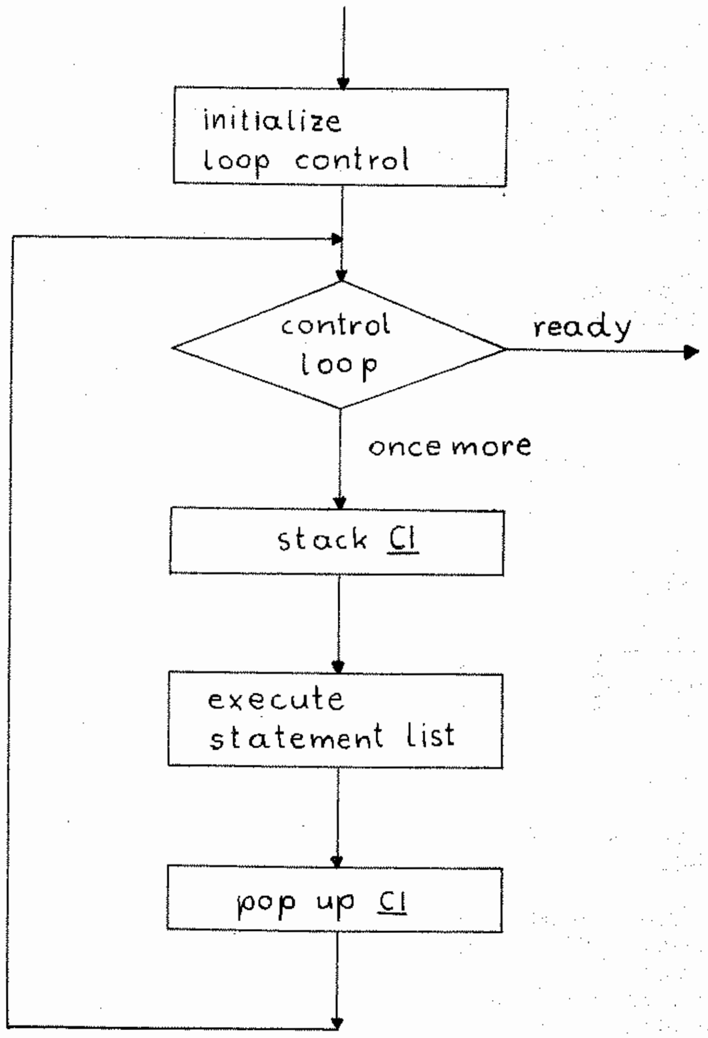


Fig. 9.8 Execution of a group

9.6 THE GOTO STATEMENT

Corresponding section of /5/:

6.4 The goto statement

A goto statement consists essentially of a reference, which refers either to a statement label constant, or to a scalar label variable, or to a function returning a label. The evaluation of this reference yields in all three cases a unique name n which, applied to the denotation directory DN, gives access to the denotation of a statement label constant: either the referenced label constant itself, or the one assigned last to the referenced label variable, or the one returned by the referenced function.

The denotation of a statement label constant consists of two components identifying uniquely the statement denoted by the label:

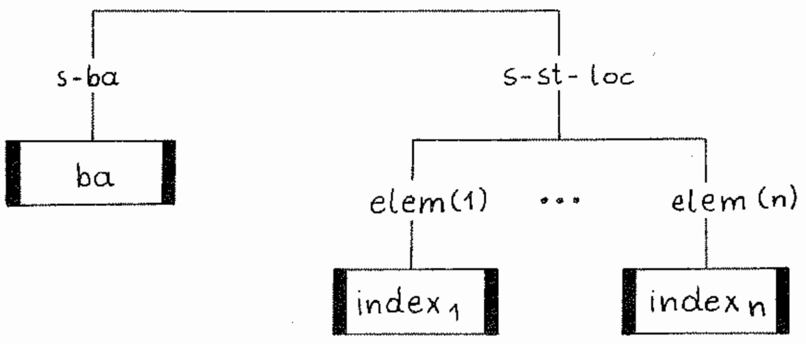


Fig. 9.9 Denotation of a statement label

(1) The block activation name ba of that block activation in which the label was declared.

(2) An index list giving the statement location of the statement denoted by the label relative to the outermost statement list of the block activation identified by ba. The localization of a statement relative to a statement list is described in 9.4. The index list was produced by the translator from the position of the label in the concrete text and inserted as declaration of the label constant into the abstract program (cf. 2.2).

The aim of a goto statement is to simulate the normal flow of control to the target statement denoted by the label, i.e., to transform the PL/I machine into that state in which it would have been if the target statement would have been encountered normally. This means first to re-establish the block activation identified by the block activation name of the label denotation, and second, within that block activation, to modify the control information in such a way, that the sequence of its contained indices, ordered from bottom to top, is the index list of the label denotation.

This is performed in four steps:

(1) The block activations established in the state, as represented by the dump D̲, are terminated one after the other by the normal block epilogue (cf. 8.2.4), until the block activation is re-established in which the label was declared, i.e., until the current block activation name B̲A̲ is the blcok activation name of the label denotation. If this block activation is not encountered up to the outermost block activation of the task, the program is in error.

(2) In the block activation re-established by step 1, the nested statement levels are terminted one after the other, until the target statement is contained (possibly nested) within the innermost not yet terminated statement list or if-statement. This is done by popping up the control information C̲I̲ (cf. 9.4), level by level, until the sequence of indices contained in C̲I̲ (except the current index) is equal to an initial portion of the index list of the label denotation. (At the latest, this is the case when all but the outermost levels of statements in the block activation have been terminated).

(3) The statement lists and if-statements containing the target statement are entered level by level until the innermost is reached. This is performed for each level by:

(a) changing the current index of C̲I̲ to the value given by the corresponding place in the index list of the label denotation,

(b) stacking the control information C̲I̲ for one level and entering into the text of C̲I̲ the statement, from the old text, which is denoted by the index just changed. This statement has to be a statement list or if-statement if the program is not in error. In particular it cannot be a group (a goto into a group is forbidden, since in such a case no loop control would have been established in the stacked control, and therefore the flow of control would not find its way out of the group correctly).

These two actions are repeated until all levels given by the index list of the label denotation are established.

(4) Finally, the current index is adjusted, i.e., set to the last value of the index list of the label denotation, and the normal flow of control is continued.

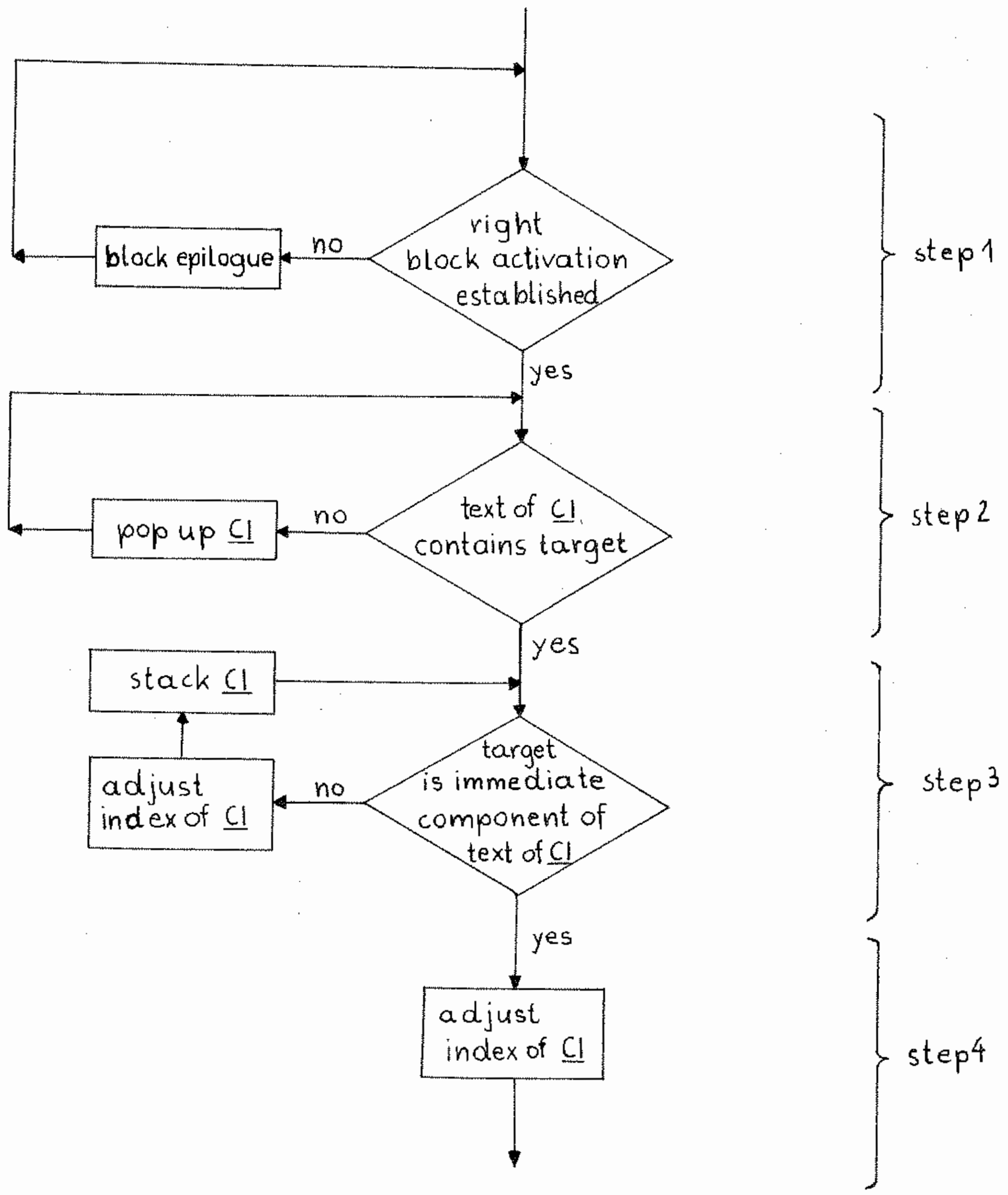


Fig. 9.10 Execution of the goto statement

Example:

```
BEGIN;
      L1 : ...;
      L2 : DO I = 1 TO N;
           L21 : IF ... THEN
                 L21T : BEGIN; ... ; GOTO L231; ... ;END;
           L22 : ...;
           L23 : DO;
                 L231 : target-statement;
                 L232 : ...;
                 END;
           END;
END;
```

In this example the denotation of the label in the goto statement consists of the block activation name of the outer block and the index list <2,3,1>. The goto statement is performed in the following four steps:

(1) The inner block activation is terminated and the outer one reestablished. Then the re-established control information CI is as shown in Fig. 9.11 (writing the labels instead of the labeled statements as text components).

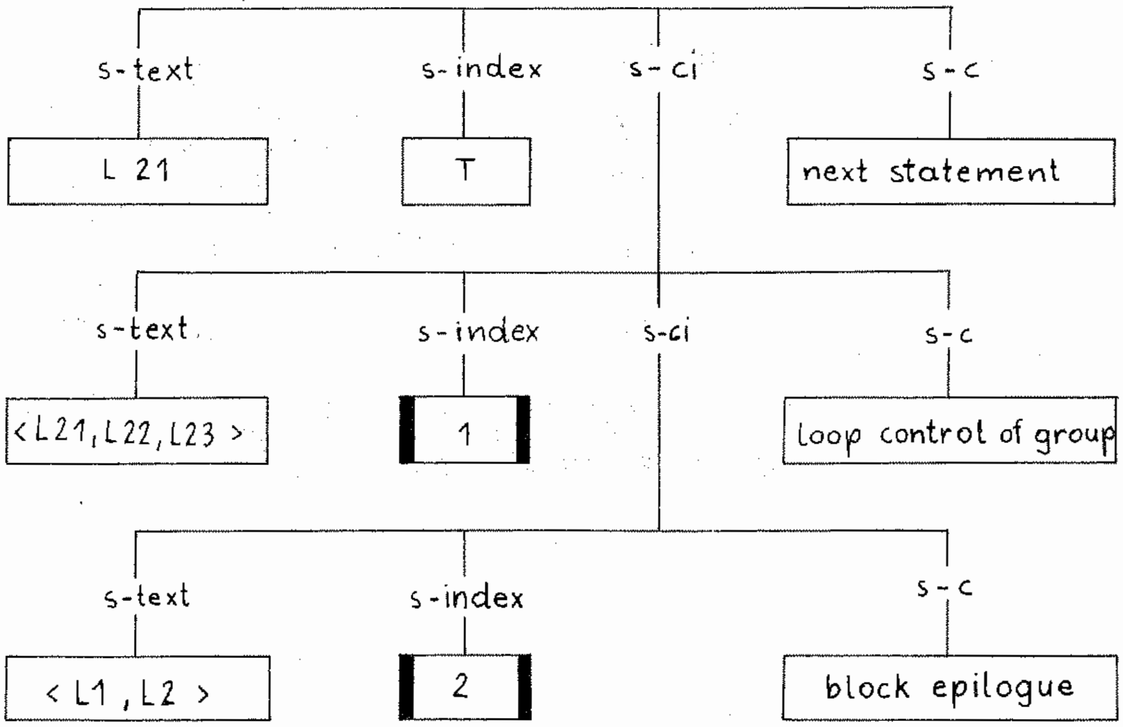


Fig. 9.11 Control information during goto example

(2) This control information is popped up one level.

(3) The index of the new control information is changed from 1 to 3 and CI is stacked one level. Then the control information is as shown in Fig. 9.12.

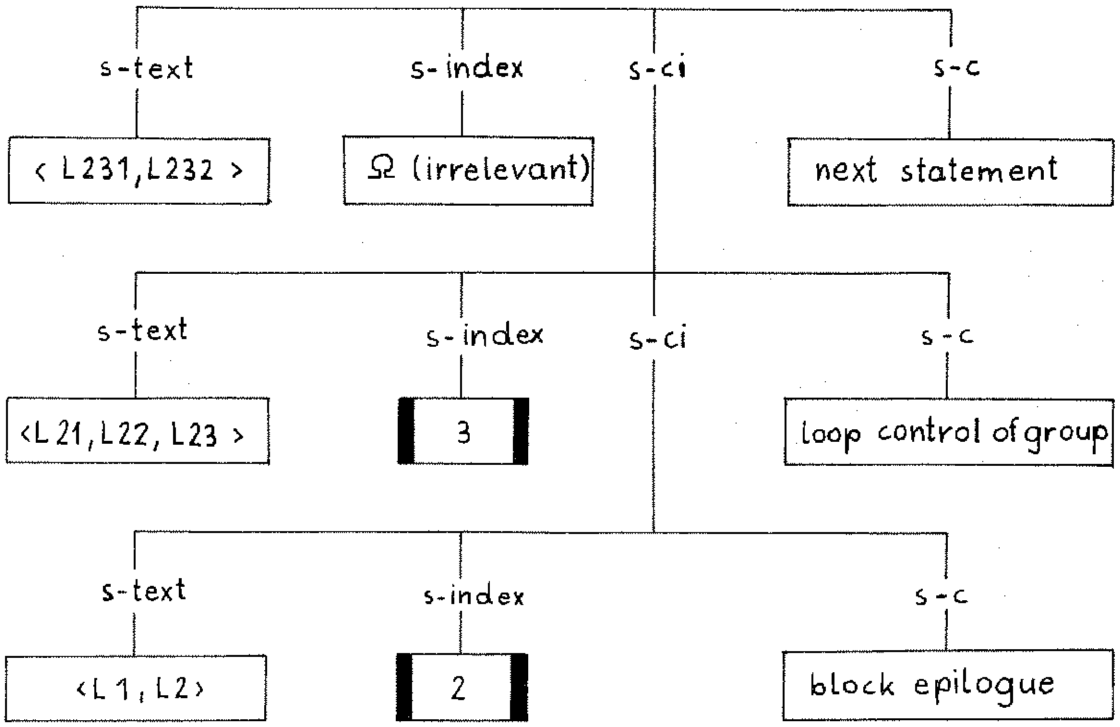


Fig. 9.12 Control information during goto example

(4) The index is adjusted to 1, and the flow of control continues normally.

It should be mentioned that in special cases one or more of these steps may be skipped. In particular, in the simplest case of a goto, namely a goto within the current statement list, only step 4 is applicable.

Example:

```
DO; L1 : ...;
    L2 : ...;
    L3 : ...;
    L4 : GOTO L2;
    L5 : ...;
END;
```

In this example only step 4 is performed, changing only the current index of CI from 4 to 2 and continuing.

10. ALLOCATION, ASSIGNMENT AND EXPRESSION EVALUATION

Corresponding sections of /5/:

7. Allocation, initialization, and freeing of variables

8. Assignment statement, expression evaluation, reference to variables

9. Data, operations and conversions

10.1 ALLOCATE STATEMENT AND FREE STATEMENT

Corresponding section of /5/:

7. Allocation, initialization, and freeing of variables

10.1.1 THE ALLOCATE STATEMENT

Corresponding section of /5/:

7.1 The allocate statement

An allocate statement specifies a list of allocations. Fig. 10.1 shows the structure of the specification of a single allocation, with an indication as to which components are significant for which types of allocation.

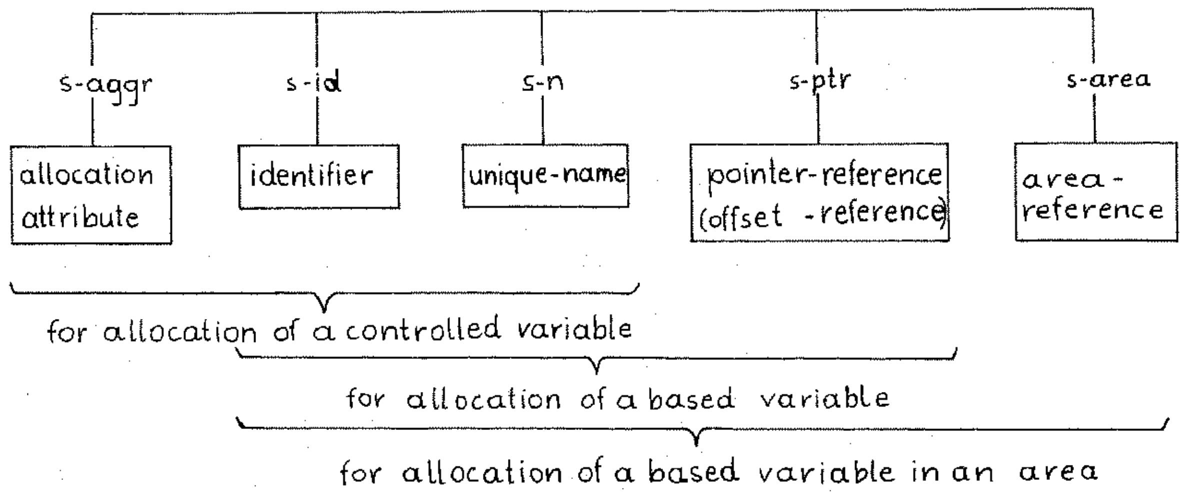

s-aggr
s-id
s-n
s-ptr
s-area
allocation attribute
identifier
unique-name
pointer-reference (offset -reference)
area-reference
for allocation of a controlled variable
for allocation of a based variable
for allocation of a based variable in an area

Fig. 10.1 Structure of specification of an allocation

The identifier is the identifier of the variable to be allocated, the unique name is the unique name of this variable. Three types of allocation have to be distinguished.

10.1.1.1 Order of execution of allocations

For determining the order of execution dependencies between the allocations specified in the allocate statement are observed.

An allocation is dependent on another allocation of a variable specified in the same statement if this variable is of controlled storage class and if any of the evaluations implied by the allocation refers to this controlled variable, and if this controlled variable itself is not yet allocated. By implied evaluation there is to be understood the evaluation of the aggregate attribute of the variable to be allocated, the evaluation of the pointer and the area reference (for based variables), and the evaluation of the initial attribute.

All allocations which are not dependent on others in the above sense, may be executed in any order. After the execution of each individual allocation the non-dependent allocations are re-determined and a selection is made. It is an error if all specified allocations are dependent on one another.

10.1.1.2 Allocation of controlled variables

The allocation of a controlled variable proceeds in the following steps:

(1) The aggregate attribute to be used for allocation is evaluated. This aggregate attribute is the one which is declared for the variable, but array bounds, string lengths, and area sizes may stem from various sources. The following rules hold:

(a) if no allocation attribute is specified in the allocate statement, then the declared aggregate attribute is evaluated.

(b) if an allocation attribute is specified in the allocate statement, then extents are taken:

from the attribute in the allocate statement if specified there by an expression,

from the aggregate attribute of the current generation of the variable if specified in the statement by an asterisk,

from the declared aggregate attribute if left unspecified in the statement.

(2) A pointer identifying the storage to be given to the variable is determined from the evaluated aggregate attribute and the main storage S (cf. 4.2.3). This pointer is added to the allocation state of S by the elementary allocation function (cf. 4.2.3).

(3) A new generation is formed from the pointer and the evaluated aggregate attribute. This generation is put on top of the list of generations associated with the unique name of the variable in the aggregate directory AG (cf. 5.3).

(4) The variable is initialized using the generation just defined, the allocation attribute specified in the allocate statement, and the aggregate attribute declared for the variable. The allocation attribute as well as the aggregate attribute of the variable may contain initial attributes in their scalar attribute components (cf. 2.2.1). If an initial attribute is given in the allocate statement, it overrides an initial attribute in the declared aggregate attribute at the corresponding position. If no initial attribute is given in the statement, the one in the declared attribute is taken.

Each initial attribute refers to a scalar component or to an array component of the variable. The assignments to be performed according to an initial attribute are determined by this attribute and the sub-generation of the variable to which it refers.

There are three different kinds of initial attributes:

(a) If the initial attribute is a nested list of expressions with replication factors, the expressions are evaluated and the replication factors evaluated and applied, the result being a list of operands. The operands are assigned successively to the scalar components of the generation to which the initial attribute refers. The process stops if the operand list is exhausted, or if the scalar parts of the generation are exhausted.

(b) If the initial attribute is a call statement, the call is performed.

(c) If the initial attribute is a list of "special initial elements", assignments of entry or label constants are performed in the following way. Each initial element specifies a subscript list and the identifier of a label or entry constant. For each of these elements, the subscript list is used to determine a sub-generation of the generation to which the initial attribute refers, the unique name associated with the specified identifier is used to form an entry or label operand, and the operand is assigned to the sub-generation.

10.1.1.3 Allocation of based variables in main storage

The pointer (offset) reference may or may not be specified in the statement. The allocation proceeds in the following steps:

(1) The aggregate attribute of the based variable is evaluated. Since the extents of a based variable may be specified only by constants or REFER-options, this means that the expressions specified in the REFER-options are evaluated and converted to integer values. The result is an evaluated aggregate attribute.

(2) A pointer is determined from the evaluated aggregate attribute and the main storage <u>S</u>. This pointer is added to the allocation state of <u>S</u>, using the elementary allocation function (cf. 4.2.3).

The pointer is also added to the free-set of the current task. The free-set is a set of pointers identifying storage parts that have been used for allocation via based variables in <u>S</u>. It is used for freeing all this storage at termination of the task.

(3) If a pointer (offset) reference is given in the statement, the generation associated with the reference is evaluated. If not, the generation associated with the pointer (offset) reference given in the declaration of the based variable is evaluated. A pointer operand is formed from the pointer determined in step (2), and assigned to the generation just evaluated.

If the generation is an offset generation (i.e., if an offset reference was specified instead of a pointer reference), the pointer is converted to an offset before assignment, using the area declared with the offset reference for the conversion (cf. 4.2.7).

(4) All components of the variable mentioned as targets in REFER-options, are assigned the values of the extent expressions specified in the respective REFER-options. These values are obtained from the extents in the evaluated aggregate attribute evaluated in step (1). Steps (3) and (4) may be done in any order.

(5) A generation is formed from the evaluated aggregate attribute and the pointer. This generation is used for initialization of the based variable. The initialization proceeds as described for controlled variables, except that no initial attributes are specified in the statement. Only initial attributes specified in the aggregate attribute of the variable are used.

10.1.1.4 Allocation of based variables in areas

The pointer and the area reference may or may not be present in the statement. The allocation proceeds in the following steps:

(1) If an area reference is specified in the allocate statement, the generation associated with this reference is evaluated. If not, the reference used for setting the pointer must be an offset reference (it is either given explicitly in the statement, or in the declaration of the controlled variable, see step (4)). The area reference given in the corresponding offset declaration is evaluated in this case, and taken as the generation of the area in which the allocation is made.

(2) The aggregate attribute of the based variable is evaluated as in step (1) of the allocation of a based variable in main storage.

(3) An offset is determined from the evaluated aggregate attribute and the allocation state of the area identified by the area generation evaluated in step (1). A test is made whether the allocation in the area is possible (the test is implementation-defined). If the allocation is not possible, the AREA condition is raised.

If the allocation is possible, the offset is added to the allocation state of the area.

(4) The pointer (offset) reference is determined and evaluated as in step (3) of the allocation of a based variable in main storage. The offset determined in step (3) is used to form an offset operand, which is assigned to the pointer (offset) generation.

If the generation is a pointer generation, the offset is converted to a pointer using the area in which the allocation is made (cf. 4.2.7).

(5) Initialization of components mentioned in REFER-options is done as in step (4) of the allocation of a based variable in main storage.

Steps (4) and (5) may be done in any order.

(6) The offset is converted to a pointer using the area in which the allocation is made (cf. 4.2.7). This pointer and the evaluated aggregate attribute are used to form a generation. The generation is used for initialization, which proceeds as for allocations in main storage.

On normal return from the on-unit called by the AREA condition, the allocation is retried after reevaluation of the area generation.

10.1.2 THE FREE STATEMENT

Corresponding section of /5/:

7.5 The free statement

A free statement specifies a list of freeings which are executed in any order. Fig. 10.2 shows the structure of the specification of a single freeing with the indication as to which components are significant for which type of freeing:

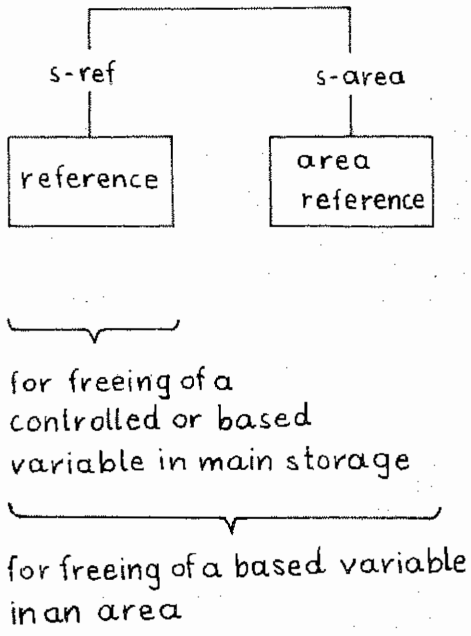

s-ref
s-area
reference
area reference
for freeing of a controlled or based variable in main storage
for freeing of a based variable in an area

Fig. 10.2 Structure of specification of a freeing

The reference is a level 1 reference to the variable to be freed.

10.1.2.1 Freeing of controlled variables

The following actions are taken:

(1) If the list of generations associated with the controlled variable in the aggregate directory AG contains the null generation only, no action is taken. If the list contains just one generation, then this generation belongs to another task, and the freeing is erroneous. In all other cases, the top generation of the list of generations is deleted.

(2) The pointer contained in the pointer part of the deleted generation is deleted from the allocation state of the main storage S.

10.1.2.2 Freeing of based variables in main storage

The following actions are taken:

(1) The generation associated with the reference of the based variable is evaluated.

(2) The pointer contained in the generation is deleted from the allocation state of the main storage. It is also deleted from the free-set of the current task, thus preventing any attempt to free the associated storage part a second time at task termination. A test is made whether the pointer was actually present in the allocation state and in the free-set.

10.1.2.3 Freeing of based variables in areas

The area reference may or may not be present in the statement. The following actions are taken:

(1) If the area reference is present in the statement, its associated generation is evaluated. If it is not present in the statement, it is obtained from the pointer qualification of the based variable reference in the following way. If the pointer qualification is a reference to the POINTER built-in function, the area reference contained in its second argument is taken. In all other cases the qualifier must be an offset reference, and the area reference specified in the offset declaration is taken and evaluated.

(2) The generation associated with the based variable reference is evaluated.

(3) The pointer contained in the generation is converted to an offset using the area identified by the area generation obtained in step (1), and this offset is deleted from the allocation state of the area.

A test is made whether the offset was actually present in the allocation state of the area.

10.2 ASSIGNMENT STATEMENT, EXPRESSION EVALUATION, REFERENCE TO VARIABLES

Corresponding section of /5/:

8. Assignment statement, expression evaluation, reference to variables

An assignment statement is specified by an abstract text consisting of a left-part, which is a list of references, and a right-part, which is an expression.

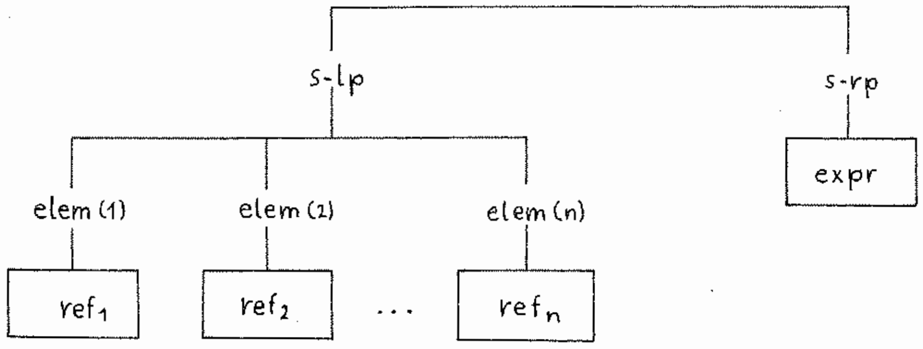


Fig. 10.3 Structure of an assignment statement

The references in the left-part are references to variables and/or to pseudo variables (pseudo variables are discussed in 13.2).

In order to simplify discussion, the term aggregate attribute of the reference to a variable will mean the aggregate attribute of the referred-to part of the variable (cf. 4.2.4). We shall also say that a reference is an array, or a structure, etc., if the aggregate attribute of the reference is an array or a structure attribute, etc.

If the references in the left-part of an assignment statement are non-scalar), the assignment statement is an aggregate assignment statement. An aggregate assignment statement is not interpreted immediately, but it is expanded into a sequence of scalar assignment statements which are interpreted sequentially.

The expansion and interpretation is governed by the evaluated aggregate attributes of the references in the left-part. (If a reference happens to be the reference to a pseudo variable, the aggregate attribute of the first argument is taken).

The evaluated aggregate attribute of a based reference in general cannot be obtained before the evaluation of the REFER-options specifying its extents, i.e., not before the evaluation of the pointer qualification (cf. 10.2.5.3). All references in the assignment statement which are expanded therefore undergo a pre-evaluation before the proper interpretation of the statement. The pre-evaluation fixes the generation of the referenced variable for the subsequent interpretation.

10.2.1 PRE-EVALUATION OF EXPRESSIONS

Corresponding section of /5/:

8.2.1.1 Pre-evaluation of expressions

The pre-evaluation of an expression concerns all those references in the expression which are expanded in the expansion of aggregate assignment statements. It does not concern, therefore, arguments of user-defined functions, and non-expanding arguments of built-in functions (whether an argument expression of a built-in function is expanded ot not is a property of the built-in function).

The pre-evaluation of the reference to a variable results in a generation. This generation is inserted as s-gen-component of the reference itself, as shown in Fig. 10.4:

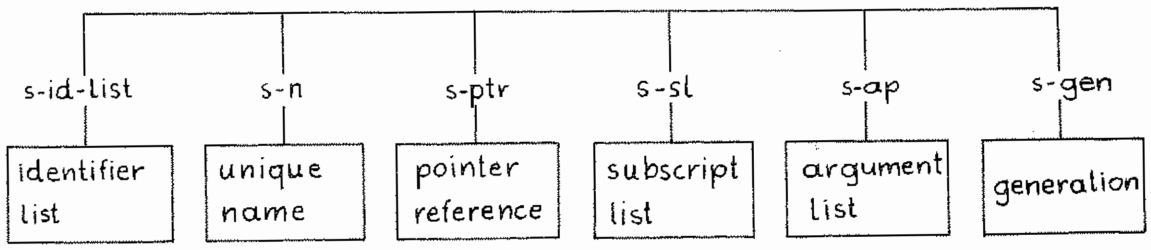


Fig. 10.4 Pre-evaluated reference

The inserted generation subsequently is used when the reference is evaluated, or when assignments to the variable are made.

For proper variables (i.e., static, automatic, or controlled variables and parameters) the current generation is obtained as the head of the list of generations associated with the variable in the aggregate directory AG. This generation is inserted as the s-gen-component of the reference. For defined variables, no action is taken in the pre-evaluation. For based variables a new generation is formed from the evaluated aggregate attribute of the based variable, and the pointer resulting from the evaluation of the pointer qualifier. The evaluation of the aggregate attribute involves the evaluation of REFER-options. The evaluation of the generation of a based variable is dealt with in 10.2.5.3.

10.2.2 EXPANSION OF AGGREGATE ASSIGNMENT STATEMENTS

Corresponding sections of /5/:

8.1.1 Expansion of aggregate assignment statements

8.1.2 Syntactic modification of expressions

As soon as the references to variables in the assignment statement have been pre-evaluated, the evaluated extents of all involved variables can be obtained from the generations associated with the references.

Let eva be the evaluated aggregate attribute of the left-most reference in the left-part. The expansion proceeds as follows:

(1) If eva is scalar, all references in the left-part must be scalar. The assignment statement is executed.

(2) If eva is non-scalar, all references in the left-part must be arrays with identical bounds, or all must be structures with an equal number of components. For each integer i which determines an immediate component of eva, proceeding sequentially from the smallest to the greatest integer, the following actions are taken:

   (a) The text of the assignment statement is modified, as determined by eva and i (see below).

   (b) This modified text is treated like the original assignment statement (this means that if the modified text specifies a scalar assignment statement it is now interpreted, otherwise it is expanded as just described).

The left-part and the right-part of an assignment statement are modified according to the same rules, the rules for modification of references being subsumed under the rules for modifying expressions.

The modification of an expression is determined by the evaluated aggregate attribute eva and the integer i and is done according to the following rules:

(1) If the expression is an infix expression, then both operand expressions are modified according to the rules for modifying expressions.

(2) If the expression is a prefix expression, then the operand expression is modified according to the rules for modifying expressions.

(3) If the expression is parenthesized, then the expression enclosed in the parentheses is modified according to the rules for modifying expressions.

(4) If the expression is a function reference, a generic reference, a label, a format label, or a scalar reference to a variable, then it is left unchanged.

(5) If the expression is an array reference to a variable and eva is an array attribute with the same number of dimensions and the same bounds, then the reference is replaced by the reference to the ith component of the array.

(6) If the expression is a structure reference to a variable and eva is a structure attribute with the same number of components, then the reference is replaced by the reference to the ith component of the structure.

(7) If the expression is a structure reference to a variable and eva is an array attribute, then the reference is left unchanged.

(8) If the expression is a reference to a built-in function, then the expanding arguments of the reference are modified according to the rules for modifying expressions. Whether an argument is expanding or not is a property of the built-in function.

(9) All cases not mentioned above are erroneous.

Assignment statements given the BY NAME option are expanded in a different way. The difference with respect to the non-BY NAME expansion arises when eva is a structure attribute. Then the information given to the modifying function is not the number of a component, but the sub-aggregate name id identifying the component in the leftmost reference of the left-part.

Rule (5) in the list of rules for modifying expressions has to be deleted for BY NAME expansion, and rule (6) has to be replaced by:

(6') If the expression is a structure reference to a variable and eva is a structure attribute, then the reference is replaced by the reference with id appended as name qualifier, provided that a component with name id of the structure exists. If no such part exists, then the modifying process containing the reference is abandoned, i.e., no assignment statement is constructed and executed in this step.

10.2.3 SCALAR ASSIGNMENT

Corresponding section of /5/:

8.1.3 Interpretation of scalar assignment statements

A scalar assignment statement is executed by

(1) evaluating the sub-generation associated with the references in the left-part (cf. 4.2.6 and 10.2.5), in order from left to right. If the references are references to pseudo-variables, the evaluation results in pseudo-generations (assignment to pseudo-variables is described in 13.2).

(2) evaluating the right-part expression, which results in an operand (cf. 10.2.4).

(3) assigning the operand to the storage parts identified by the evaluated generations, in order from left to right.

The assignment of an operand implies the conversion of the operand, using the data attribute of the generation as target data attribute (cf. 10.3.2). The conversion process also contains a check as to whether the assignment is at all possible and the raising of conditions.

Let the converted operand be op' and the pointer part of the generation be p then a new storage part S' is created on assignment (cf. 4.2.2):

$\underline{S}' = \text{el-ass}(\text{s-vr}(op'),p,\underline{S})$

10.2.4 EXPRESSION EVALUATION

Corresponding sections of /5/:

8.2.2 Evaluation of expressions in entry-context

8.2.3 Evaluation of expressions in non-entry context

The evaluation of an expression results in an operand (cf. 4.1.1). In general it must be known whether an expression is evaluated in a context expecting an entry operand or not. This is the case if the expression is the right-hand side of an assignment statement whose left-hand references refer to entry variables, or if the expression is in an argument specified as entry. It is said in this case that the expression is in entry context.

10.2.4.1 Evaluation of expressions in entry context

An expression in entry context may be one of the following:

(1) A parenthesized entry expression. The expression enclosed in parentheses is evaluated according to the present rules, which gives the result of the expression.

(2) The reference to an entry constant. If no argument list is specified in the argument part of the reference, an operand is formed from the attribute ENTRY and from the value representation resulting from the representation of the unique name of the entry (cf. 4.1.2). The operand is passed as the result of the expression. If argument lists are specified in the argument part, the entry is called as a function, using the first argument list in the argument part, the denotation of the entry name , and the entry attributes (parameter description and return type) of the entry name. The operand returned by the function call must be an entry operand. If there is no argument list left in the argument part, this operand is passed as result. If there are argument lists left, the value of the entry operand is used to perform another function call, using the next argument list of the argument part and the entry attributes given in the return type of the original entry. The resulting operand is again treated as just described.

(3) A reference to an entry variable. The reference to the variable is evaluated, giving an entry operand. This operand is treated as

described in (2), i.e., function calls are performed until all argument lists in the argument part of the reference are used up.

(4) A reference to a generic name. If there is no argument list specified in the argument part of the reference and the reference is in an argument position to a procedure, the generic selection is made using the parameter descriptor list of the corresponding entry parameter description. If an argument list is specified, the attributes of the arguments are used for performing the generic selection. The result of the selection is an entry expression (cf. 8.3.4). This entry expression is evaluated according to the present rules, giving an entry operand.

The entry operand is treated as described in (2), i.e., function calls are performed until all argument lists in the argument part are used up.

(5) A float-generic builtin function. This is possible only if the reference is in an argument position of a procedure. The generic selection is performed using the parameter descriptor list of the corresponding entry parameter description (this selection is not further described in this informal document).

10.2.4.2 Evaluation of expressions is non-entry context.

The following are the possible forms of an expression and the rules for their evaluation in non-entry context. An expression may be:

(1) An infix expression. Both operand expressions are evaluated in any order according to the rules for evaluating expressions. The operator subsequently is applied to the two resulting operands, giving the result of the expression (cf. 10.3).

(2) A prefix expression. The operand expression is evaluated, the operator is applied subsequently to the resulting operand, giving the result of the expression (cf. 10.3).

(3) A parenthesized expression. The expression enclosed in parentheses is evaluated, which gives the result of the expression.

(4) A constant. A constant in the abstract text consists of a data attribute and a value. It is converted to the form of an operand, using the value representation resulting from representing the value of the constant (cf. 4.1.2). The operand is passed as the result of the expression.

(5) A reference to a variable. References to variables are described in 10.2.5. If the reference is to an entry variable, the resulting entry operand is used to perform a function call. The function call is performed with the first argument list of the argument part of the reference. If no argument list is specified, the empty argument list is assumed. If the function again returns an entry operand, it is again treated as just described, i.e., it is used for another function call.

(6) A reference to an entry constant. The entry is called as a function with the first argument list specified in the argument part of the refernce. If no argument list is specified, the empty argument list is assumed. If the resulting operand happens to be

an entry operand, it is called again with the next argument list in the argument part. A resulting non-entry operand is passed as the result of the evaluation.

(7) A reference to a generic name. The generic selection is made using the attributes of the arguments in the first argument list of the argument part (cf. 8.3.4). The selection yields an entry expression, which is evaluated according to the rules for evaluation in entry context (cf. 10.2.4.1). The resulting operand is used for performing a function call as described in (6).

(8) A label, format, or file constant. An operand is formed from the data attribute LABEL (for label and format constants) or FILE (for file constants) and the value representation resulting from representing the unique name of the constant (cf. 4.1.2). The operand is passed as the result.

(9) An isub-variable. Isub-variables occur only in connection with isub-defining (cf. 10.2.5.2.1). The integer value attached to the variable is used to form an integer operand, which is passed as the result.

10.2.5 REFERENCE TO VARIABLES

Corresponding section of /5/:

8.3 Evaluation of references to variables

The evaluation of the reference to a variable proceeds in the following steps:

(1) The generation currently associated with the variable is evaluated. For proper and based variables this evaluation is done <u>already</u> during the <u>pre</u>-evaluation (cf. 10.2.1).

(2) The subscript-expressions occurring in the reference are evaluated and converted to integer values in order from left to right, the sub-aggregate names (name qualifiers) occurring in the reference are replaced by the integer values identifying the respective sub-aggregate (cf. 4.2.4). This step results in a reference list (cf. 4.2.6).

(3) The sub-generation determined by the generation and the reference list is evaluated (cf. 4.2.6).

(4) The operand determined by the (scalar) sub-generation is evaluated.

Step (2) also includes checking whether the subscripts are within the range given by the evaluated aggregate attribute of the variable. If a subscript is outside the range, the SUBSCRIPTRANGE condition is called (if enabled). The evaluated data attribute of a variable is obtained:

(a) for proper variables (i.e., STATIC, AUTOMATIC, or CONTROLLED storage class and parameters) from the aggregate attribute part of the generation of the variable, which for proper variables is immediately accessible,

(b) for defined variables from the denotation of these variables (cf. 5.4),

(c) for based variables by evaluating the REFER-options contained in the aggregate attribute (cf. 10.2.5.3).

Proper variables, defined variables, and based variables differ in step (1),i.e., in the way the generation associated with the variables is obtained. Step (1) is discussed separately for these variables below.

The reference process does not include step (4) if the reference is in the left-part of an assignment statement, or if it is the argument to a procedure and the generation of the argument is to be passed to the corresponding parameter (cf. 8.3.1). The sub-generation resulting from step (3) need not be scalar in the last case.

10.2.5.1 Proper variables

The generation currently associated with a proper variable, if existing, is obtainable as the head of the list of generations contained in the aggregate directory AG under the aggregate name associated with the variable. The aggregate name is obtained as the entry made in the denotation directory DN under the unique name of the variable (see the diagram in 5.3).

No generation exists if the variable has not been allocated.

10.2.5.2 Defined variables

The declaration of a defined variable specifies an aggregate attribute, optionally a position (which is singificant for overlay defining only), and a reference which is called the reference to the base variable, or base reference. Three kinds of defining must be distinguished.

10.2.5.2.1 Isub-defining

A defined variable is isub-defined if no position is specified and if the base reference contains subscript expressions which contain at least one reference to an isub-variable. A reference to an isub-variable is syntactically distinguishable from references to all other kinds of variables, and is characterized by an integer value. An isub-defined variable is always an array variable.

The reference to an isub-defined variable proceeds in the following way (this covers steps (1) and (3) of the above general scheme):

(1) Let d be the number of dimensions of the defined array variable. Then the first d elements of the evaluated reference list (see step (2) above) are used to give the values to d isub-variables (those characterized by the integers 1 to d). These values are inserted as s-v-components of the references to the isub-variables in the subscript expressions of the base reference.

(2) The generation associated with the base reference is evaluated.

(3) A consistency check is made between the aggregate attribute of the elements of the defined array variable, and the aggregate attribute as given in the above sub-generation. The same conditions must be satisfied as in the simple-defined case between

attributes of the defined variable and the generation associated with the base reference.

(4) The rest of the reference list, not used to give values to isub-variables, is used to determine the sub-generation of the above sub-generation.

Note that if a reference to an isub-variable occurs in an expression the operand resulting on evaluation is formed immediately from the value found in the s-v-component of the isub-reference.

Example:

Let D and B be declared as:

```
DCL 1 D (2,2) DEFINED B (2SUB, 1SUB), 2 X, 2 Y,
    1 B (2,2), 2 U, 2 V;
```

and consider the reference D (2,1) . Y

The evaluated reference list is <2,1,2>.

The number of dimensions d of the defined array variable is d = 2, so there are 2 isub-variables, which get associated with the first two elements of the reference list:

1SUB ... 2

2SUB ... 1

The reference B (2SUB,1SUB) is now evaluated. The relevant reference list is <1,2>. Let gen be the generation associated with the variable B, then we obtain the sub-generation gen' determined by gen and <1,2>.

The rest of the first reference list of the reference to the defined variable (not used to give values to isub-variables) is <2>. The final result is the subgeneration gen'' determined by gen' and <2>.

10.2.5.2.2 Simple defining

A defined variable is simple-defined if no position is specified in the declaration, and if its evaluated aggregate attribute is equal to the aggregate attribute of the base reference, disregarding array bounds and string lengths. Corresponding array bounds must be such that the bounds in the base array comprise the bounds in the defined array, string lengths in the base array must be shorter than or equal to corresponding string lengths in the defined array.

The evaluation of the generation associated with the defined variable proceeds in the following way:

(1) the base reference is evaluated , giving the corresponding sub-generation,

(2) the aggregate attribute in the aggregate attribute part of this generation is replaced by the evaluated aggregate attribute of the defined variable.

The new aggregate attribute part in the generation specifies which parts of the storage associated with the base generation can be used by the defined variable. This modified generation may be non-connected, even if the base generation is connected. The modified generation is the input to step (3) in the general scheme in 10.2.5.

Example:

Let D and B be declared as

DCL D (2,2) DEFINED B, B (2,3);

Let p be the pointer part of the generation associated with B. Fig. 8.4 symbolically shows the storage p(S) associated with B and the storage corresponding to parts of B, in a linear model. It also shows the storage usable by D, which is a non-connected part of S (cf. 4.2.5).

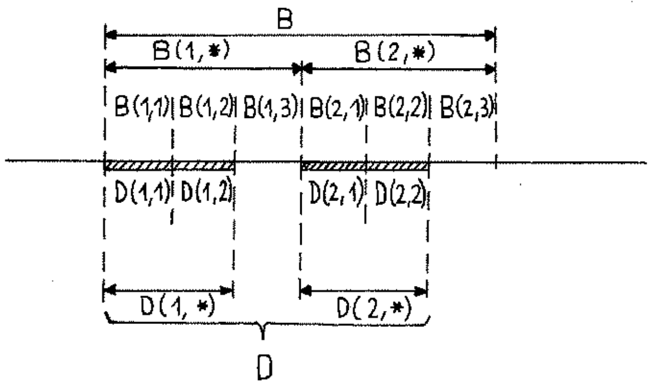


Fig. 10.5 Example for simple defining

10.2.5.2.3 Overlay defining

A defined variable is overlay-defined if it is an aggregate of unaligned strings, and if the base reference is a string aggregate of the same type (BIT or CHARACTER), and if the condition for simple defining is not satisfied. Overlay defining has to be assumed in any case if a position is specified.

The number of elements (bits or characters) in the base reference minus the specified position must not be smaller than the number of elements in the defined variable minus 1.

For describing overlay defining the term linear index of an element of an aggregate is introduced, where element here means a single bit or character. The mapping function introduces a left to right ordering of the immediate components of aggregates (cf. 4.2.4) and thus a tree structure with ordered branches for a whole aggregate. The linear index of an element which is at a terminal node of the tree, is its position number obtained by counting the terminal nodes from left to right.

The evaluation of the generation associated with the defined variable proceeds in the following way:

(1) the generation associated with the base reference is evaluated; this generation must be connected,

(2) a new pointer is found which identifies that storage part which is associated with the ith up to the jth element of the base reference, where i is the integer specified by the position attribute, and j - i + 1 is the number of elements in the defined variable,

(3) a new connected generation is formed using the evaluated aggregate attribute of the defined variable, and the new pointer.

It is a property of the storage mapping function that a pointer of the required properties always can be found, and that the storage part now associated with, say, the kth element of the defined variable is exactly the storage part associated with the (i + k - 1)th element of the base reference (cf. 4.2.7). The new generation is the input to step (3) in the general scheme in 10.2.5.

Example:

Let D and B be declared as

```
DCL D(3) BIT(1) UNALIGNED DEFINED B POS(2),
    1 B UNALIGNED,2 X BIT(2), 2 Y BIT(3);
```

Fig. 10.6 symbolically shows the storage associated with B and its parts, and the corresponding parts of D.

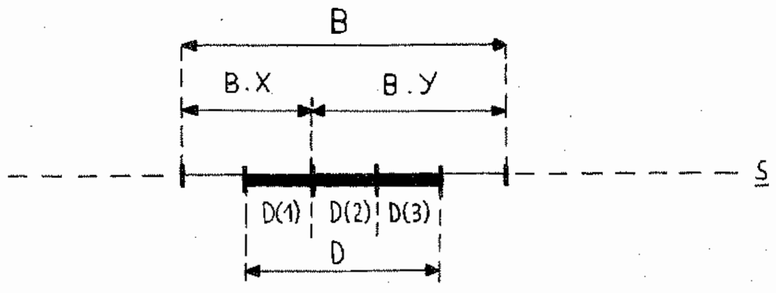


Fig. 10.6 Example for overlay defining

10.2.5.3 Based variables

The declaration of a based variable specifies an aggregate attribute and optionally a pointer reference. Extents are either specified by constants or by REFER-options. The reference to a based variable may be pointer qualified, i.e., the reference, besides identifier list and argument list specifies a pointer reference. The evaluation of the generation associated with the reference to a based variable proceeds in the following way:

(1) If the reference is pointer qualified, the qualifying pointer reference is evaluated. If it is not qualified, the pointer reference specified in the declaration of the based variable is

evaluated. The result is an operand which specifies a pointer value.

(2) The REFER-options in the aggregate attribute are evaluated. Each REFER-option specifies an expression and an identifier list. The expression is used for initialization after the allocation of a based variable (cf. 10.1.1.3). The identifier list identifies a scalar structure component within the based variable. This component, within the storage identified by the above pointer, contains the value which is used as the current extent for which the REFER-option stands.

In order to retrieve this value, storage identified by the above pointer is accessed. For this purpose an intermediate generation is built from the pointer and the aggregate attribute evaluated up to the point (from left to right) containing the REFER-option. REFER-options therefore are evaluated from left to right, if this order is relevant, in any order otherwise. The operand resulting from the storage access via the intermediate generation is converted to an integer value.

(3) If all REFER-options are evaluated, the resulting evaluated aggregate attribute and the pointer evaluated in (1) are used to construct a generation. This generation serves as input to step (3) in the general scheme in 10.2.5.

It is only for certain cases that the resulting generation is a sensible means for referencing storage. Let eva-b be the evaluated aggregate attribute of the based variable and let the storage part identified by the pointer value be originally associated with a variable (or part of a variable) with aggregate attribute eva-p. If eva-p is scalar, then the value representation associated with the storage part makes sense together with eva-b only if eva-p and eva-b are equal, no relationships between value representations associated with non equal attributes being defined. If eva-p is non-scalar, then the meaningful parts of the storage part are identified by the storage mapping function map(eva-p,i), the storage parts identifiable with the based variable, however, are given by map(eva-b,i), and again no relationship is defined beween the values of the mapping function for non equal arguments.

There is an exception to the general rule that eva-b and eva-p have to be equal, which is called the <u>left-to-right equivalence</u> rule. This exception is due to a property of the mapping function which is guaranteed by the language. If eva is a structure attribute, then the location of a structure component depends only on the properties of eva up to (from left to right) and including that component. Consequently if eva-b and eva-p are structure attributes, a reference to a structure component of the based variable gives defined results if eva-b and eva-p are equal up to that component.

10.3 INFIX AND PREFIX OPERATIONS, CONVERSION, NUMERIC PICTURES

Corresponding sections of /5/:

9.3 Evaluation of infix expressions

9.4 Evaluation of prefix expressions

9.5 Conversion

9.6 Pictures

10.3.1 INFIX AND PREFIX OPERATIONS

The part of expression evaluation which is to be described in this section is the application of infix or prefix operators to already evaluated, but not yet converted operands; these operands are objects consisting of an evaluated aggregate attribute part and a value representation part, as described in 4.1. The result of the operation is again an operand, and whereas the aggregate attribute part of this result operand is completely defined by the language (except for the implementation-defined maximum and minimum precision associated with arithmetic data attributes), the value represented by the value representation part is generally not so defined. For character string comparison, this aspect is treated by introducing an implementation-defined collating function; hence the main problem was to characterize the operations on numerical values and pointers in a way which treats accurately certain subcases without defining the rest, and this was solved by postulating suitable axioms.

First, appropriate target attributes are computed and the operands are converted to these targets. These target attributes depend only on the data attributes of the operands, except for the case of fixed-point exponentiation (in which target and result attributes depend also on the value of the second operand). It is convenient here that the target for a conversion may be an incomplete attribute (cf. 10.3.2). For example, for arithmetic infix operators, the common target for conversion of the two operands is the object shown in Fig 10.7,

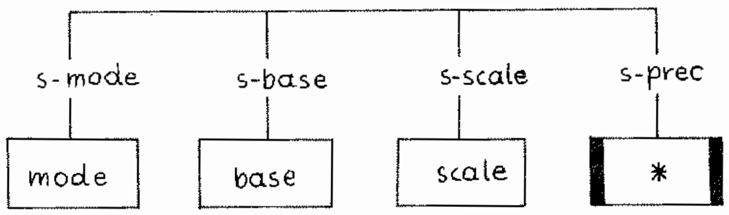


Fig. 10.7 Target attribute for conversion during arithmetic infix operations.

where mode, base, and scale are the higher of the respective characteristics of the attributes of the two operands; for the arithmetic prefix operators PLUS and MINUS, the target is the object AR-EDA (cf. 10.3.2). In the first case, the precision (and where necessary, the scale factor), in the second, all characteristics are obtained by means of the convert-1-instruction from the incomplete target and the source attribute.

For an infix operation, let eva_1, eda_1 and vr_1 be the aggregate attribute, the data attribute and the value representation of the converted first operand, eva_2, eda_2 and vr_2 those of the second. The data attribute eda-res_0 of the result operand is computed as a function of eda_1 and eda_2. To obtain the value representation vr-res_0 of the result operand, the converted operands are transformed into their values v_1 = value(eva_1,vr_1), v_2 = value(eva_2,vr_2), and a result value v-res_0 is computed from v_1 and v_2 (depending, possibly, also on eda_1 and eda_2); then vr-res is obtained as the representation of v-res with eva-res_0 consisting of eda-res_0 and the corresponding default density.

For arithmetic operators, the first step is the transition from v_1 and v_2 to v-res_0 is a test, whether the operator is applicable; if not, the ERROR or ZDIV condition is raised, otherwise, v-res_0 is computed. The operation to be applied to v_1 and v_2 is not guaranteed in general to be the exact mathematical operation corresponding to the operator, but may be an implementation-dependent approximation thereof, whose accuracy may depend on eda_1, eda_2. However, the following is postulated:

> If v_1 and v_2 belong to the sets v-0-set(eda_1) and v-0-set(eda_2) of values which are guaranteed to be exactly representable with eda_1 and eda_2, respectively (cf. 4.1.2), then, in case of fixed-point eda_1 and eda_2 and an operator which is not division, the result v-res_0 will be the exact mathematical result.

__Example:__

> If eda_1 is REAL DEC FIXED (3,0), eda_2 is REAL DEC FIXED (4,1), v_1 is 237, v_2 is 844.2, and the operator is PLUS, then v-res_0 is 1081.2; if v_2 were 844.25, then v-res_0 = 1081.25 would not be guaranteed.

Before v-res_0 is represented with eda-res_0, a test for overflow or underflow is made. This test is very similar to that for the SIZE condition (cf. 4.1.2), except that instead of the precision of eda-res_0, the maximum precision associated with eda-res_0 is used. The following can be derived from the definition of eda-res_0 and the axioms for v-res_0:

> If v-res_0 is guaranteed to be the exact mathematical result, and if additionally no FIXED OVERFLOW situation arises, then v-res_0 is in the v-0-set(eda-res_0), i.e., is guaranteed to be exactly representable with eda-res_0.

__Example:__

> In the example given above, if the maximum precision for real decimal fixed attribute is at least 5, then the result attribute eda-res_0 will be REAL DEC FIXED(5,1), no overflow will occur, and the result value v-res_0 = 1081.2 will indeed be exactly representable with eda-res_0.

For comparison operators, the numeric case is treated axiomatically with similar postulates; the character or bit string case, like the string operators on the whole, and the file and the label cases [1] present

1) It should be noted, however, that in the file and label case not the values themselves, but the denotations accessible by means of these values are compared (cf. 9-17(77) of /5/).

no dificulties; The pointer case is again treated axiomatically, with the following postulates (for the operator EQ;NE is defined as negation of EQ):

(1) If the two pointers are the same, then EQ yields true.

(2) If the two pointers are independent (cf. 4.2), then EQ yields false.

For prefix operators, the general sequence of steps is the same, though the details are much simpler. For the prefix operator MINUS, a test for overflow or underflow must be made, because the predicates testing for them are not necessarily invariant against change of sign (e.g., an implementation may use asymmetric two-complement representation for binary numbers).

10.3.2 CONVERSION

Conversion is performed by an instruction <u>convert-1</u>(eva-tg,op) which has as arguments the target attribute eva-tg and an operand op which is to be converted to this target. The result is the converted operand. The eda-part, eda-tg, of the target eva-tg may be incomplete. If so, it is completed (see below). Except in the case of area conversion (which is treated at the end of this section), conversion to a complete target eva-tg falls into three steps:

(1) The operand op is transformed into a value.

(2) The value is converted into a value of the type determined by eda-tg.

(3) The converted value is represented with eva-tg; the result of conversion is the operand whose evaluated aggregate attribute is eva-tg and whose value representation is the obtained representation.

The first and the third step are performed by the function value(eva,vr) and by the instruction <u>test-rep</u>(eva,v), as described in 4.1.2. (The instruction <u>test-rep</u> rather than the function rep is necessary, because SIZE, STRZ or CONVERSION conditions may be raised).

The second step, called <u>value conversion</u>, distinguishes between the different types of values, e.g., numeric, character string, etc. (cf. Fig. 4.1). Conversion is only possible if the source and the target are either of the same type or if each of them is of one of the types numeric, character string, or bit string. For identical source and target type, the second step is the identity operation, and the third step may be the inverse of the first.

<u>Examples</u>:

(1) If the source attribute eda-op (the eda-part of eva-op) and the target attribute eda-tg are both arithmetic, then step 1 yields a numeric value which is left unchanged by step 2 and transformed back into an operand by step 3; if eda-op and eda-tg are the same, then this operand will be op under certain additional restrictions (cf. 4.1.2).

(2) If eda-op is a bit string attribute and eda-tg a binary picture attribute, then step 1 will yield a bit string value, step 2 a numeric value, step 3 its representation in pictured form.

(3) If eda-op is a numeric picture attribute, eda-tg a character string attribute, then step 1 will yield a numeric value which, with the aid of eda-op, is transformed by step 2 into a character string value; the representation of this character string value will be, under certain restrictions, the same as the value representation part of the original operand op.

(4) If eda-op and eda-tg are both numeric picture attributes, then the numeric value computed by step 1 will be transformed back into pictured form. Even if eda-op and eda-tg are the same, this will not under all circumstances be guaranteed to be the unchanged original representation, because un-normalized floating-point representations (produced by overlay-defining) will be normalized.

The definition of the operation of value conversion distinguishes between the six possible combinations of different source and target types. In <u>numeric to character conversion</u>, the source attribute is needed; if it is numeric picture, then this is essentially the operation of representing a numeric value in pictured form (cf. 10.3.3); if it is arithmetic, then again a picture attribute is constructed, though it differs somewhat from the treatment of the ordinary picture case. Also in <u>numeric to bit conversion</u>, the source attribute is needed. In <u>character to numeric conversion</u> a scan from left to right is made, and at each stage a test is made as to whether a correct continuation of the string is still possible; if not, the CONVERSION condition is raised; the method by which this test is made is that developed for stream input transmission. In <u>character to bit conversion</u>, the target attribute is needed, because only as many characters as necessary are converted (and hence can raise the CONVERSION condition). The two remaining cases, <u>bit to numeric conversion</u> and <u>bit to character conversion</u>, present no problems.

Area conversion is accomplished by the instruction <u>area-conv</u>(eva,op), where op is the operand to be converted and eva is the target attribute. An area operand is constructed whose vr-part has the size which corresponds to eva, the same allocation state as the vr-part of op, and which in the parts identified by the allocation state is identical with the vr-part of op.

Before the actual conversion a test is made whether the conversion is possible. The conversion is possible if by a sequence of allocations a value representation of the size corresponding to eva can be given the allocation state of the vr-part of op. If the conversion is not possible the AREA condition is raised.

As was said above, the target attribute presented as first argument to the <u>convert-1</u>-instruction may be incomplete; that means, any component may be specified by *. Examples of incomplete attributes are the following objects AR-EDA and STRING-EDA (Figs. 10.8a and 10.8b):

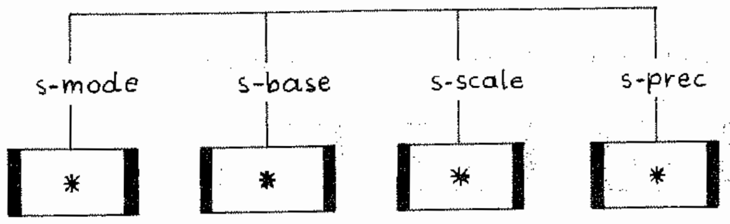


Fig. 10.8a The incomplete arithmetic attribute AR-EDA

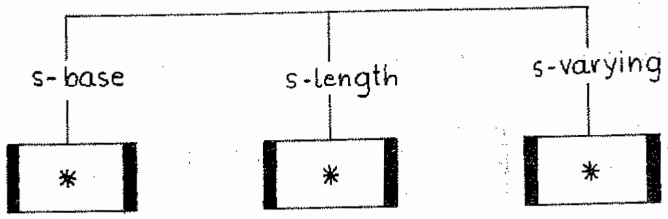


Fig. 10.8b The incomplete string attribute STRING-EDA

They can be used to specify "conversion to arithmetic" or "conversion string" without specifying particular characteristics. Another example is given in 10.3.1. The completion of the components occupied by * can be done by the convert-1-instruction with the help of the source attribute.

10.3.3 REPRESENTATION AND EVALUATION OF NUMERIC PICTURES

Only a very brief description of the concepts introduced and used in 9.6 of /5/ will be given.

We consider first the relation between picture attributes in concrete and in abstract text; the aim in choosing the particular form of abstract syntax of pictures as defined in 2.2.3 of /5/ was to make explicit as much as possible of the structure which is needed by the interpreting functions and instructions, without too much of a burden on the translator. Thus, the partition of a fixed-point picture into mantissa field and scale factor, of a floating-point picture into mantissa field, exponent separator, and exponent field is made explicit by showing these parts as different components. Also, the unit position of the mantissa and the division of sterling fields into subfields is shown by separate pointers rather than by characters in the field description.

Examples:

The three picture attributes (of mode REAL, say) which in concrete representation read '-ZZ.V9F(-3)', 'ZZ.9E99', and 'G+M99M8M7', are translated into the following abstract form (Figs. 10.9a,b,c):

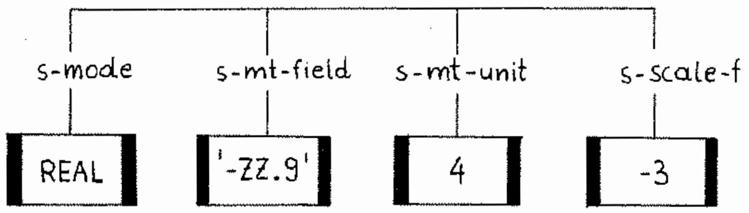


Fig. 10.9a Decimal fixed-point picture

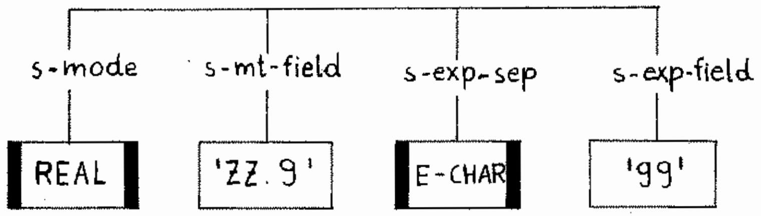


Fig. 10.9b Decimal floating point picture

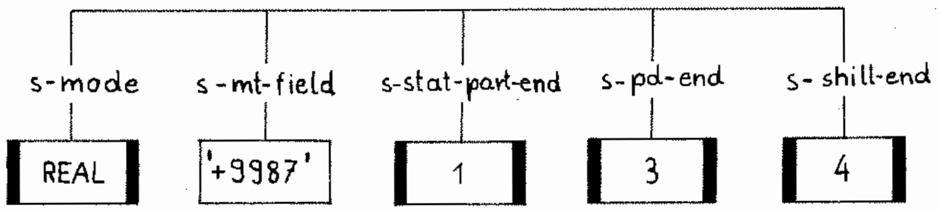


Fig. 10.9c Sterling picture

In these figures, strings have been represented by their concrete equivalents. Certain picture characters are not translated into their immediately corresponding abstract characters: so, S becomes SIGN, H becomes S-CHAR, P becomes D-CHAR.

Next, we introduce <u>explicit picture attributes</u>. These differ from the picture attributes of the abstract syntax in that zero-suppression or drifting information, where present, is given a more explicit form, the subfield description is transformed into the corresponding unsuppressed form, and explicit components containing the drifting information are added.

<u>Examples</u>:

(1) For a subfield description '$ZZ.9', the unsuppressed form is '$99.9', the explicit form is shown by Fig. 10.10a (this time strings being presented in their abstract form):

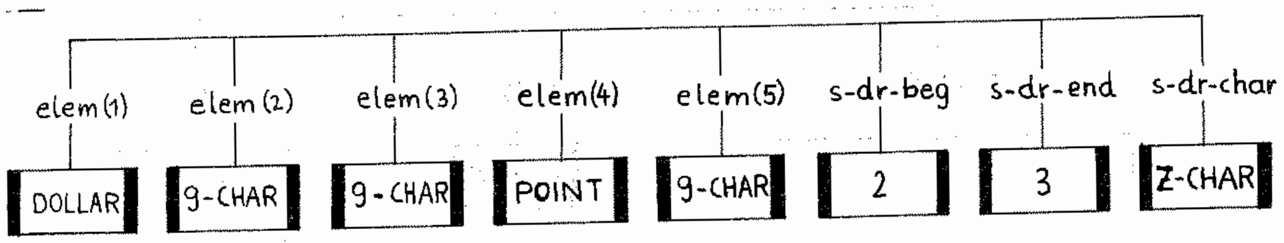


Fig. 10.10a Subfield of explicit picture attribute

(2) For a subfield description '$,$$$' the unsupressed form is '$,999', the explicit form is shown by Fig. 10.10b:

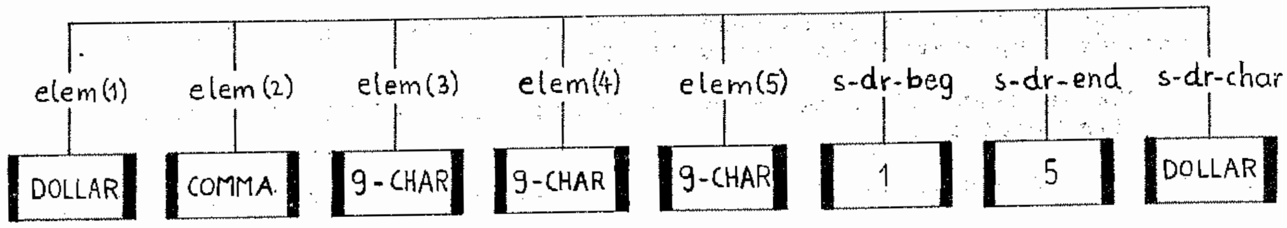


Fig. 10.10b Subfield of explicit picture attribute

The essential step in the representation of a numeric value with a numeric picture data attribute is the transformation of the numeric value into a string value. As an intermediate step in this transformation, the concept of pictured value is used. A <u>pictured value</u> has the same structure as a picture attribute (in explicit form), but the picture specification characters may be replaced by other characters; e.g., the characters 9-CHAR may be replaced by the digits of the number to be represented. In fact, the representation of a subfield consists in writing digits, sign characters, etc., as they come from the numeric value, into the picture attribute; only as a last step, the finally resulting pictured value is "linearized" to a string value.

<u>Examples</u>:

To represent 0,123 with a picture attribute which in concrete form reads '999ES9', the following pictured value is constructed from 0.123 and the abstract form of the picture attribute (Fig. 10.11):

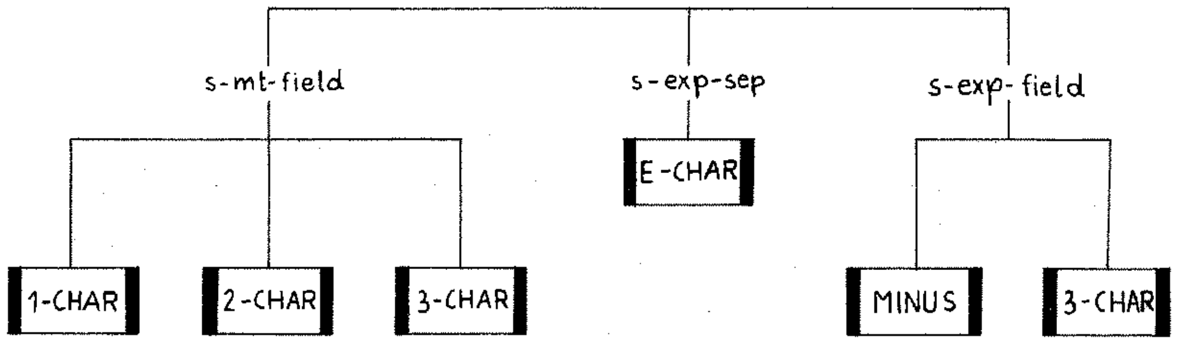


Fig. 10.11 Pictures value

This pictured value is then linearized to the string value which, in concrete representation, reads '123E-3'.

The sequence of steps in representing a subfield is (for decimal and sterling pictures):

(1) The number to be represented is decomposed into a number list.

(2) The elements of this list are transformed into characters; this may include e.g. overpunching. The characters are written into the appropriate positions of the pictured value.

(3) The sign is represented (if not treated already in step 2).

(4) Zero suppression or drifting is performed, if specified. (Steps 1-3 will have used the unsuppressed form of the picture attribute.)

A test for the SIZE condition is included.

The process of retrieving a numeric value from its representation in pictured form is, in the main, defined implicitly as the inverse of the representation process. Since conditions may be raised, the definition is given by an instruction; also, certain "normalization rules" must be postulated because there may be different values with the same representation.

11. ATTENTIONS AND CONDITIONS

Corresponding sections of /5/:

3.5 State components for attentions and conditions

10. Attentions and conditions

The following abbreviations are used in this section:

abn-ret	information for abnormal return
AN	attention directory
attn-identifier	attention identifier
cap	condition action part
cbif	condition builtin function
cond	condition
cond-bif-part	condition builtin function part
CS	condition status
D	dump
eattn-cond	evaluated attention condition
EI	epilogue information
EN	attention enabling state
enable	element of the enable-list
EV	attention environment directory
id	identifier
ident	attention identification
info-list	attention information stack
intg-val	integer value
n	unique name
pref-part	prefix part
ptr-val	pointer value
ref	reference

st	statement
tn	task name

11.1 STATE COMPONENTS FOR ATTENTIONS AND CONDITIONS

The attention directory AN, the attention enabling state EN and the attention environment directory EV are solely used to describe the enabling and disabling mechanism of attentions in the various tasks and to stack the attention information.

The condition status CS contains the enabling information for prefix controlled conditions. The other two parts of CS, holding information for interpreting on-units and condition builtin functions, are used for all conditions including the attention conditions.

11.1.1 ATTENTION DIRECTORY AN

The major state component dealing with the interpretation of an attention is the attention directory AN. An attention is installed in AN by an enable statement.

A single attention is found in AN by the attention identification, a selector characterizing an attention (cf. 5.10), which is the major part of the evaluated attention condition (cf. 11.1.5 Fig. 11.4). For each attention identification four components are contained in AN:

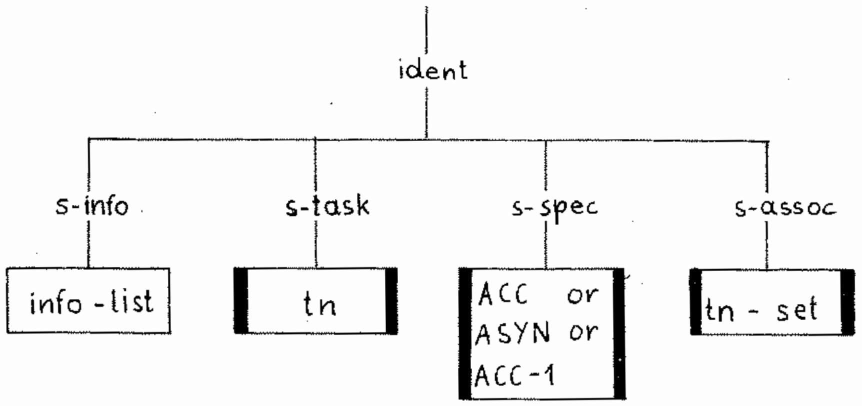


Fig. 11.1 A single attention of AN

The s-info component contains the attention information which is created with an attention occurrence and stacked in this component. This attention information is used by an asynchronous attention interrupt or by an access statement.

The s-task component names the task in which the attention is enabled at the moment.

The s-spec component characterizes the enabling mode relevant for interpretation.

The last component collects the names of the tasks with which this attention is associated.

The entries in the last three components may be changed by enable or disable statements, or by task termination.

11.1.2 THE ATTENTION ENABLING STATE EN

To handle the enabling and disabling correctly each task has a state component EN, the attention enabling state, which indicates whether attentions are enabled or only associated with a task. This information is kept in the first two components of EN, which contain evaluated attention conditions (cf. 11.1.5 Fig. 11.4).

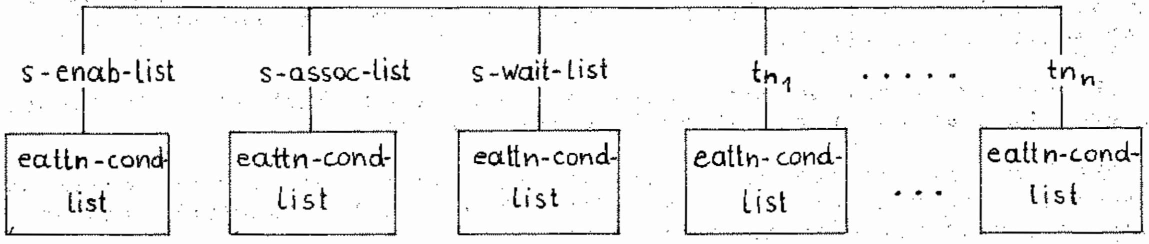


Fig. 11.2 Attention enabling state EN

In the list containing associated attentions the evaluated attention condition has the two non-empty auxiliary components which are needed when the attention becomes enabled for this task.

The component selected by s-wait-list enumerates the attentions which are only associated with the task but which are not specified with an event, so that the task must wait until all these attentions have been enabled for the task.

The last component consists of a set of event names and contains, for each event name, the corresponding evaluated attention conditions which are only associated with the task.

11.1.3 THE ATTENTION ENVIRONMENT DIRECTORY EV

The attention environment directory EV contains, for each unique name of an attention identifier (cf. 5.10), an evaluated environment used to create the attention identification (cf. 11.1.1).

11.1.4 THE CONDITION STATE CS

The major state component dealing with the interpretation of condition situations is the condition state CS. The condition state CS is a block local state component and consists of four major parts.

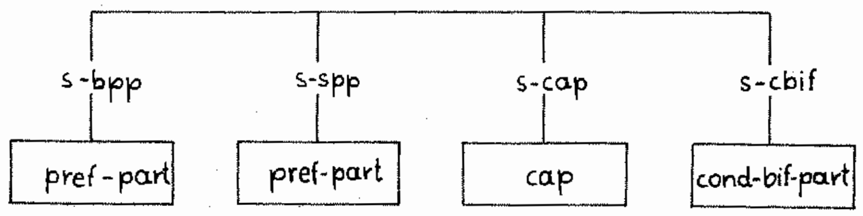


Fig. 11.3 Condition state CS

The block prefix part (selected by s-bpp), and the statement prefix part (selected by s-spp), control the condition enabling status for all conditions which may be prefixed to blocks or procedures and statements, respectively.

The condition action part contains the actions which are established by executing an on-statement or a revert statement.

The last part is the condition builtin function part, which contains components for the values of every condition builtin function, and some auxiliary ones for obtaining these values.

11.1.5 EVALUATED CONDITIONS AND CONDITION SELECTORS

The conditions appearing in the various condition and attention handling statements are evaluated before interpretation. The evaluated condition differs from the description in the abstract syntax only with respect to three conditions: the evaluated check condition is a reference to a base element; the evaluated I/O-condition has instead of a ref component the file name; and the evaluated attention condition consists mainly of the attention identification (cf. 11.1.1).

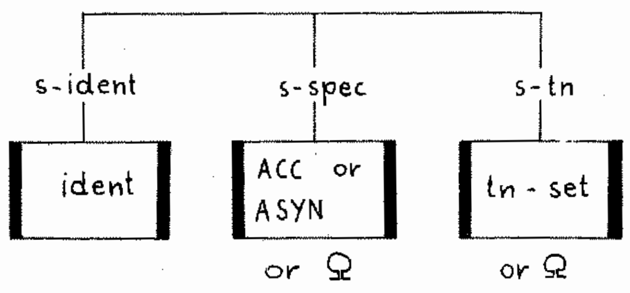


Fig. 11.4 Evaluated attention condition (eattn-cond)

To connect the condition with the proper condition action and to handle the condition prefixes correctly in CS, a condition selector is created. As identifiers appear in several conditions, a dynamic interpretation is necessary to ensure unambiguity of reference.

This is done by creating a unique selector from the evaluated condition, by connecting the unique name or the file name or the attention identification or the identifier with a simple selector. Where no unique name is needed, for example with the conversion condition, the condition selector consists only of the simple selector, e.g., s-conv. Thus a dynamic interpretation of condition prefixes is ensured.

11.2 ENABLING AND DISABLING

11.2.1 ENABLING AND DISABLING OF CONDITIONS

Several PL/I on-conditions can be enabled or disabled under control of condition prefixes. At the beginning of the program interpretation a standard enabling status exists. This is reflected in the block prefix part of the initial state of CS. This status may be modified by prefixes in front of blocks, procedures, or statements.

Condition prefixes control the enabling and disabling in a static scope. As identifiers may appear in several prefixes, a dynamic interpretation of prefixes is necessary to ensure unambiguity of reference. This is done by using the condition selector for entries in the prefix part of CS.

As condition prefixes heading a begin block or a procedure statement have the scopes of the respective blocks, they are interpreted at block entry or at procedure entry respectively (cf. 8.2). The updating of the condition enabling status of a block or procedure is done by merging the evaluated condition prefixes of the statically encompassing block with the prefixes explicitly specified for the block or procedure.

During the interpretation of PL/I statements the enabling status as defined by the block prefix part of CS can be modified by explicit statement prefixes (cf. 9.1). A similar merging is done, and the resulting enabling status of the statement is kept in the statement prefix part. The statement prefix part is only valid for the specific statement and is never stacked.

11.2.2 ENABLING AND DISABLING OF ATTENTIONS

11.2.2.1 Enable statement

The interpretation distinguishes between enabling with or without event option. When the event-option is specified for one element of the enable-list (cf. Fig. 11.5), the event generation is evaluated and the attention-event, essentially consisting of the event generation, is attached under a newly created unique name in PA (cf. 7.3). The sole effect of this is to hinder any subsequent assignment to the event variable.

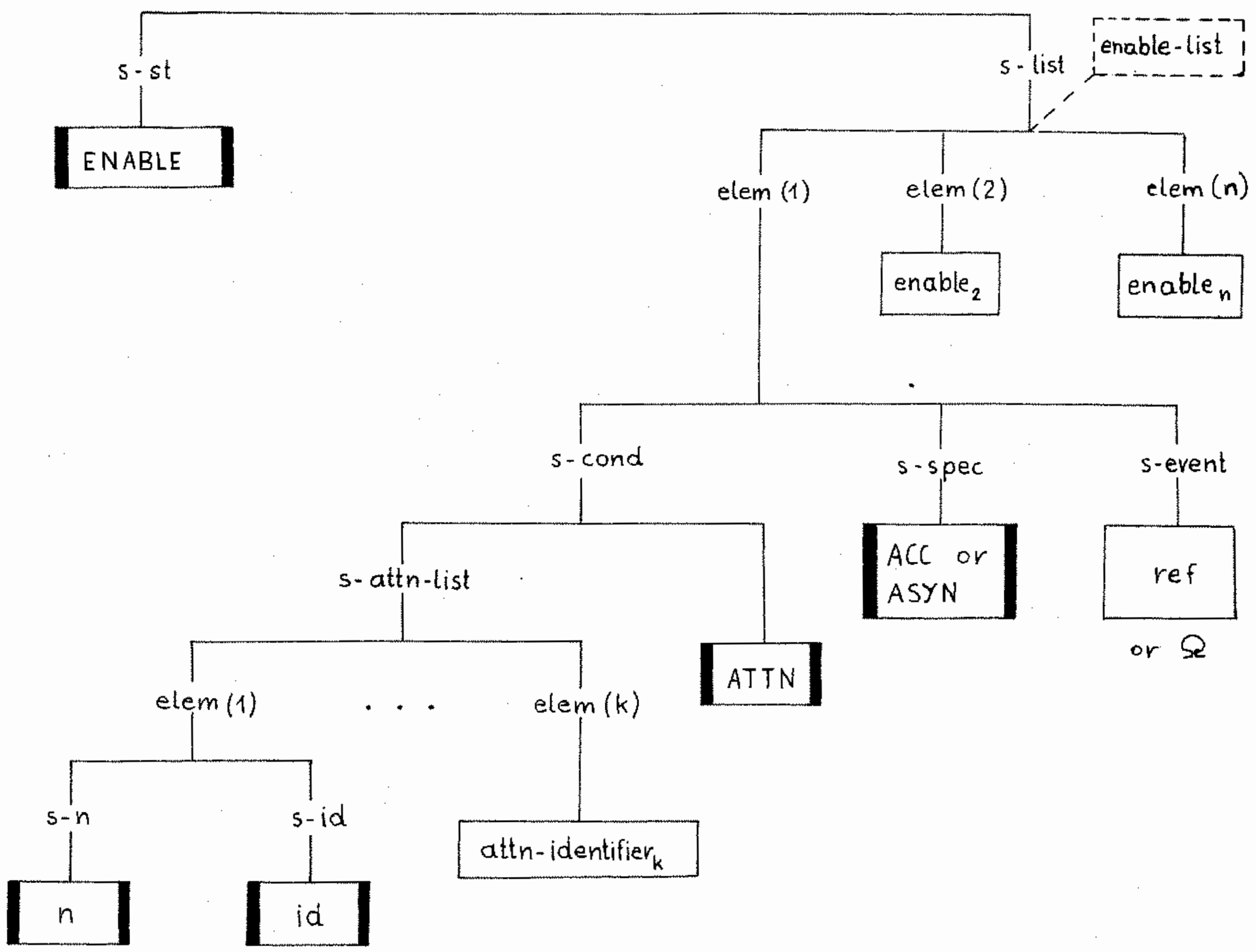


Fig. 11.5 The enable statement

The attention identifiers in each element of the enable-list are modified to attention identifications (cf. 11.1.1) and a list of evaluated attention conditions is generated for each element of the enable-list.

The attention directory AN is searched for entries for every evaluated attention condition:

(1) When no entry is found the attention is newly installed in AN, and the s-enab-list component in EN is updated with this condition.

(2) When the attention is already enabled for this task, then the enabling mode is changed to the newly specified enabling mode. If the altered enabling mode is asynchronous and the attention

information stack of this attention in AN is not empty, then an asynchronous interrupt is immediately executed.

(3) When the attention is already enabled for another task, it is only possible to associate this attention with this task. Two cases are to be distinguished:

(a) The attention is not already associated with this task:

Then the set of task names in AN is amended by the new task name. Into the list of associated attentions in EN the evaluated attention condition, (with non-empty auxiliary components), is entered.

When an event is specified, then an entry is made under the event name in EN. When it is specified without an event, then the condition is concatenated with the s-wait-list component of EN.

(b) The attention is already associated with the task:

The new evaluated attention condition replaces the condition with the corresponding attention identification in the list of associated attentions in EN.

If an event is specified the condition is also entered into EN under this event name.

When the enabling of one element of the enable-list with event-option is finished and it was possible to enable all the attentions related to this event successfully, the attached event is deleted.

After the whole enable-list is interpreted a check of the s-wait-list component of EN is made: When this list is empty, the next statement is interpreted; otherwise the task is set into the wait state, where it remains until the list is empty.

11.2.2.2 Disable statement

The disable statement disables or disassociates attentions from the task and possibly enables them in another task. The attention identifiers are evaluated, and evaluated attention conditions created, as with the enable statement. Three cases have to be distinguished with the disabling of each attention condition:

(1) When the attention is neither enabled nor associated with the task, it is ignored.

(2) When the attention is only associated with the task, the corresponding entries in AN and EN are deleted.

(3) The attention is enabled in the task:

(a) When no task requires this attention for enabling, i.e., the s-assoc component of this attention in AN is the empty set, then the attention is deleted in AN and also removed from the list of enabled attentions in EN.

(b) The task which currently enables the attention, is selected in an implementation defined manner from the set of tasks which have this attention only associated with them. The corresponding attention condition with the enabling mode is found in the s-assoc-list component in the attention enabling state EN of the selected task. The attention condition is transferred from this list to the list of the enabled attentions. The two additional components are correspondingly updated.

In AN the task name is replaced in the s-task component by the new one and also deleted from the set of task names there.

The enabling mode for the attention is possibly updated. When the enabling mode in the task, which gets this attention enabled, is asynchronous and the attention information stack in AN is not empty, an asynchronous interrupt is prepared in the selected task and all tasks are set into the active state.

In all other cases only the tasks are set into the active state.

When one of the lists, kept in EN under an event name, gets empty, the corresponding attention event is deleted from PA.

11.3 CONDITION ACTION

The standard system action is defined by the language, while the on and revert statement allows the programmer to define actions.

11.3.1 STANDARD SYSTEM ACTION

The standard system action specifies various actions for the various conditions. In most cases the error condition is raised and a comment written, while in other cases only a comment is written.

Special actions are required as standard system actions by the endpage condition, not raised by signal statement, the check condition and the error condition.

The standard system action for some conditions like the attention condition result in no action.

When the error condition is raised the condition name is passed in the cbif argument to the error condition call, to handle the updating of condition builtin functions correctly (cf. 11.6).

11.3.2 ON AND REVERT STATEMENT

An on-statement specifies an action, which will be executed when the specific condition has been raised. The interpretation of an on-statement for a specific condition establishes the new condition action, consisting mainly of the on-unit and some additional information, in the condition action part of the current condition state CS. The condition written in the on-statement is evaluated to yield a list of evaluated conditions. The condition action is then stored in CS for every element of the list.

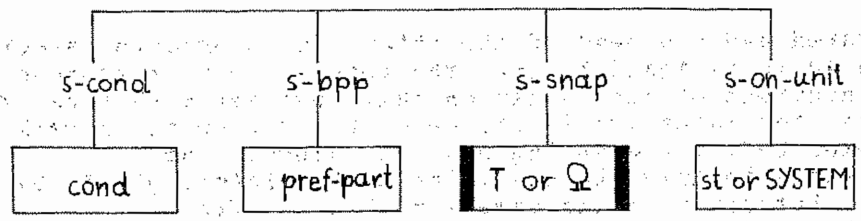


Fig. 11.6 Condition action

The on-unit is a statement, without label prefixes, which is not a group, a return statement, a procedure, an on-statement, or an if-statement.

A subsequent execution of an on-statement for the same condition in the same block, replaces the old condition action by a new one (which is taken from the executed on-statement). The condition action, when not newly specified, is inherited to all descendants of a block or procedure, and is stored there in the local condition state.

Whenever a revert statement is interpreted, the condition action of the block local condition state is deleted, and the condition action of the encompassing block is taken out of the dump D and reinstalled.

The execution of a revert statement as the only statement of an on-unit has no effect due to the activation of a new block for the interpretation of the on-unit.

11.4 ATTENTION ACTIVATION

11.4.1 ASYNCHRONOUS INTERRUPT

Attentions occur outside the PL/I machine. Such an attention consists of an attention identification part and the attention information (cf. 11.1.1). The attention directory AN is altered in the environment step (cf. 6.2) from outside in such a way that the attention information is stacked in the attention information stack of the corresponding attention. The attention identification part is identical with the attention identification of the evaluated attention condition (cf. 11.1.5 Fig. 11.4). Entries in AN are only possible, when an enable statement was previously executed and thus the attention installed in AN.

Because PL/I allows asynchronous interrupts for attentions, in each interrupt step of the computation (cf. 6.2), the attention directory AN is searched for attentions, whose attention information stack was recently altered in the environment step and whose enabling mode is asynchronous. If such an attention is found an asynchronous interrupt is executed.

The asynchronous interrupt is prepared in the task the name of which is found in AN for the specific attention. To immediatly activate the attention condition call, the task is dumped and a new block activated, which solely calls the attention condition.

The <u>attention condition call</u>, after some prepatory actions, leads to a normal condition call. These preparatory actions are listed below:

Storage is allocated and the head of the attention information stack of the attention in <u>AN</u> is assigned to it. The pointer to that storage is passed to the cbif argument of the condition call together with a newly created oncode value, for later use in the on-unit. Before the normal condition call is executed, the enabling mode is changed from asynchronous to accessed to make the condition call uninterruptable by another asynchronous interrupt of the same attention. After interpretation of the condition call the original enabling mode is reinstalled, if not changed by the on-unit, and the allocated storage is freed.

11.4.2 ACCESS STATEMENT

The access statement makes attention information available for processing from an attention, whose enabling mode is accessed.

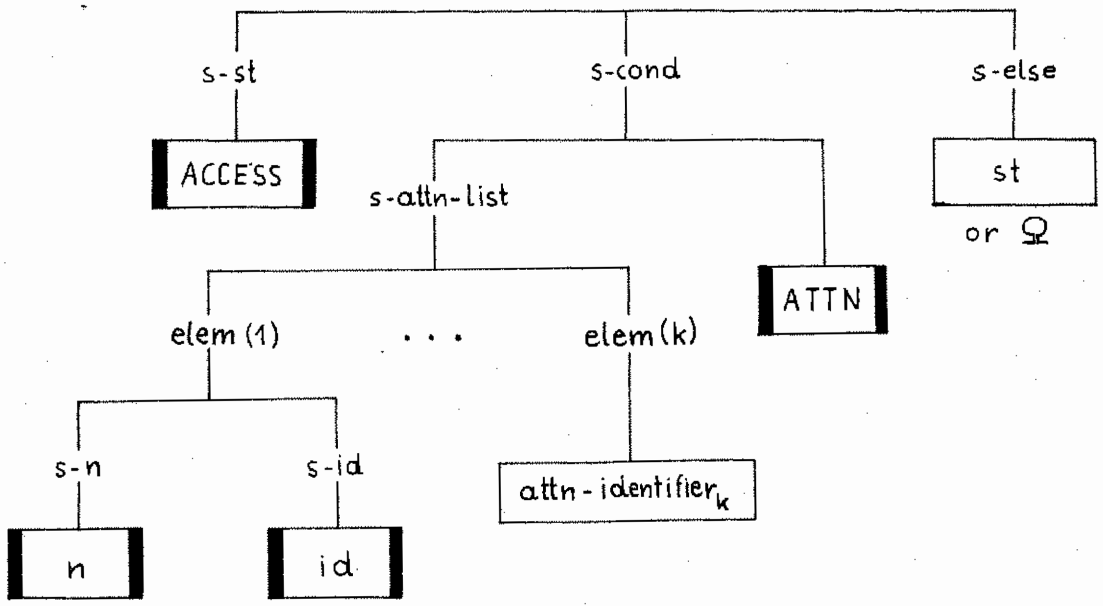


Fig. 11.7 The access statement

If the list of attention identifiers specified in the statement is empty an arbitrary attention satisfying the following condition is chosen: it must be enabled in the task with accessed mode and the attention stack must not be empty. Using this attention the attention condition call is interpreted. If no such attention exists the else-unit of the access statement is interpreted. In its absence, the task is set into the wait state as long as such an attention can not be found.

When a list of attention identifiers is specified in the access statement, a list of evaluated attention conditions is created. All these must be specified with accessed enabling mode, otherwise it is an error.

The attention condition to which the condition call is made is chosen in the following manner: The first condition is taken which is enabled in the task and whose attention information stack is not empty. If no such attention exists the interpretation is as described above with empty attention identifier list.

The attention condition call is handled analogous to that for asynchronous interrupts (cf. 11.4.1).

11.5 CONDITION ACTIVATIONS

The raising of a condition is caused either by an interrupt or by a signal statement. The execution of a signal statement for a condition causes the condition to be raised immediatly. The actual condition call is interpreted in the same way as the condition call raised by interrupt (cf. 11.4.1).

The activation of the various conditions is described in the several places in the interpreter, where they can occur (see for example raising of the check condition with the assignment statement; cf. 8-3(1) of /5/). Also the attention activation (cf. 11.4) leads to the condition call.

Special actions have to be performed for the check, the conversion and the I/O-conditions, before the general interpretation of the condition call (call-cond-1; cf. 10-19(55) of /5/). The check condition, raised with a list of references, has to be expanded, and then the call is executed for every element of the expanded reference list. The ordering of elements in the list is relevant.

If a conversion condition is not raised by a signal statement but through an actual conversion error, a specific action is activated which either allocates a dummy and passes the corresponding generation or passes the generation passed to it, to the onsource builtin function and in both cases passes an integer to the onchar builtin function. The value returned after the interpretation of the condition call may be modified through pseudovariables.

Before the call to an I/O-condition is interpreted, the values of some condition builtin functions are completed (in all cases the onfile value is set) and passed to the call.

The interpretation of the condition call must distinguish between prefix controlled and uncontrolled conditions. The statement prefix part of the condition status CS carries the information whether the prefix controlled condition is enabled or not. This information has to be tested before raising. Only the raising of the underflow condition, the check condition and the attention condition, if they are disabled, result in no action. In all other cases, if disabled conditions are raised, the program is in error. Furthermore, the condition call must distinguish the conditions which on normal return from the condition action permit further interpretation, from those conditions, which on normal return arrive at an error situation. A special action is required after the return from the call to the error condition.

The condition state is now inspected for an appropriate condition action for the specific condition, which is done through the condition selector created from the condition. If no action is present or the on-unit component of the condition action part of CS is SYSTEM, then the standard system action is executed, otherwise the condition action is interpreted. In both cases a snap action may precede it.

Condition actions are interpreted in close analogy to parameterless procedures. The current state is stacked, and a new block activation is established. The condition status is updated with the block prefix part of the corresponding on-statement, which was also reserved in the condition action. A special epilogue information is constructed and installed in EI.

After updating the condition builtin function part, the on-unit is interpreted, so that it may use the updated values of the condition builtin functions. The on-unit is interpreted like a single statement. When this is finished, the block activation is terminated. The stacked state is reinstalled and the next instruction is executed. In some cases this may lead to an error which finishes the interpretation.

11.6 CONDITION BUILTIN FUNCTION STATUS

Condition builtin functions change the value they return as a consequence of condition raising. The new value obtained remains unchanged in the dynamic descendence of the condition raising, i.e., in all blocks entered from an on-unit executed as a consequence of the condition raising. If a condition is raised and only the standard system action is executed, no change of the condition builtin function values is required. Only when the standard system action for a condition results in raising the error condition, the condition builtin functions return the same values as in an on-unit for the condition.

The information needed to interpret condition builtin functions is kept in the condition builtin function part of CS (cf. 11.1.4). This part consists of one component for each condition builtin function, and of some additional components.

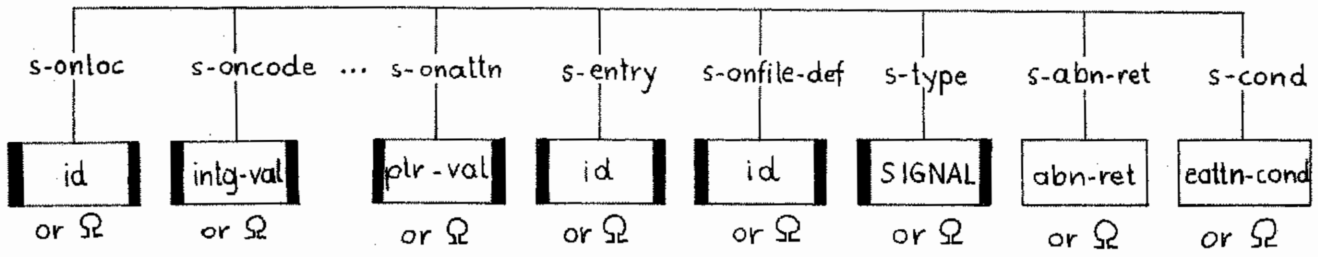


Fig. 11.8 The condition builtin function part of CS

The values of the components of the condition builtin functions in CS are updated every time a condition action different from the standard system action is interpreted. The auxiliary components selected by s-entry and s-onfile-def get their values directly in various places of the interpreter. All the other values are entered into a special argument (cbif argument) constructed at the point of the condition raising and are then passed to the condition call.

According to the various conditions, the instruction now inserts the proper values into the corresponding components. Thereby the entries in the auxiliary components are used: The value for the onloc component is taken from the s-entry component of the condition builtin function part of CS; the onfile value, in the case of the conversion condition, is taken from the s-onfile-def component.

The component selected by s-type is used to distinguish conditions raised by interrupt from conditions raised by a signal statement. The component selected by s-abn-ret is needed to provide proper completion of I/O-events after a GOTO out of an on-unit called during a wait statement. The s-cond component in CS is used to distinguish attention conditions from other conditions, and therefore allows changing the enabling mode to the mode before the condition call in the case of an abnormal return from an on-unit (cf. 11.4.1).

The value of oncode is defined by an implementation defined function dependent on the point of interrupt.

The components of the condition builtin function part may then be used by the condition builtin functions to get their values. If condition builtin functions are used out of proper context (that means outside an on-unit for the specific condition) or if the corresponding component of CS is Ω, the functions return standard values as described for the individual builtin functions.

12. INPUT AND OUTPUT

Corresponding sections of /5/:

11. Input and output

3.6 Input and output

The followimg abbreviations are used in this section:

fd,FD	file directory
FU	file union directory
ES	external storage
S	(internal) storage
M	message storage (or message part)
PA	parallel action part
TE	task-event specification
f	file name
u	file union name
env-attr	(unevaluated) environment attribute
et	evaluated statement text
ref	reference
expr	expression

This chapter describes the totality of I/O actions as summarized in Fig. 12.1. The basic concept dealing with the external storage, the association between data sets and file unions, and the logical statements about data set mapping have been outlined in section 4.3. A familarity with the notions developed there would help in reading this chapter. However, a detailed knowledge is only needed for section 12.2.3.1, and 12.5.3.

A very short description of files is given in section 5.5. It is particularly helpful for sections 12.2 and 12.3.

The structuring of this chapter follows more or less the steps in the interpretation of I/O statements. Statements which do not refer to files, i.e., stream I/O with string source or target and message I/O (Fig. 12.1), will get only peripheral mention.

I/O type:	record I/O	stream I/O			message I/O	
governed by:	record file	stream file ——	stream file standard system print file	string source or string target	display statement	certain standard system on-units
most characteristic state components involved:	FD, FU, ES			FU, S	M	
initiated or terminated by:	opening or closing			get statement or put statement		
input by:	read statement	get statement	——	get statement	reply of display statement	
output by:	locate statement or write statement	put statement copy action	put statement copy action check standard system action	put statement	display of display statement	certain standard system actions of on-conditions
update by:	read statement or rewrite statement or write statement or delete statement					

Fig. 12.1 Survey on I/O

12.1 EVALUATION OF STATEMENT OPTIONS

Corresponding sections of /5/:

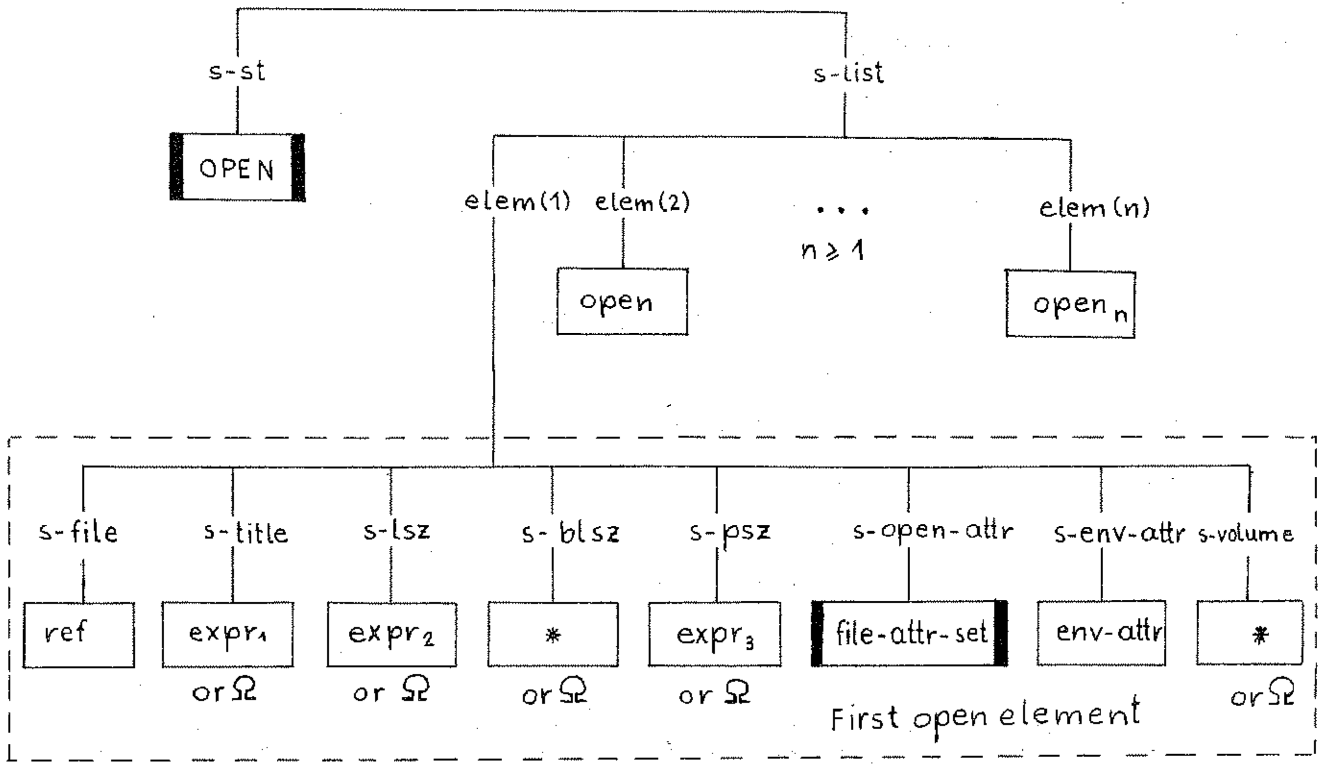
11.3.1 Open and close statement

11.4.1 Interpretation of statement options

The following abbreviations are used in this section:

lsz	linesize
blsz	type indication for lsz
psz	pagesize
ident	identification (e.g., key, display message)
idto	target identification
spec	specification

With respect to the statement text one may discriminate between open and close statements (Fig. 12.2), record I/O and display statements (e.g., into-read statement in Fig. 12.3), and stream I/O statements (e.g., file-put and string-put statement in Fig. 12.4).



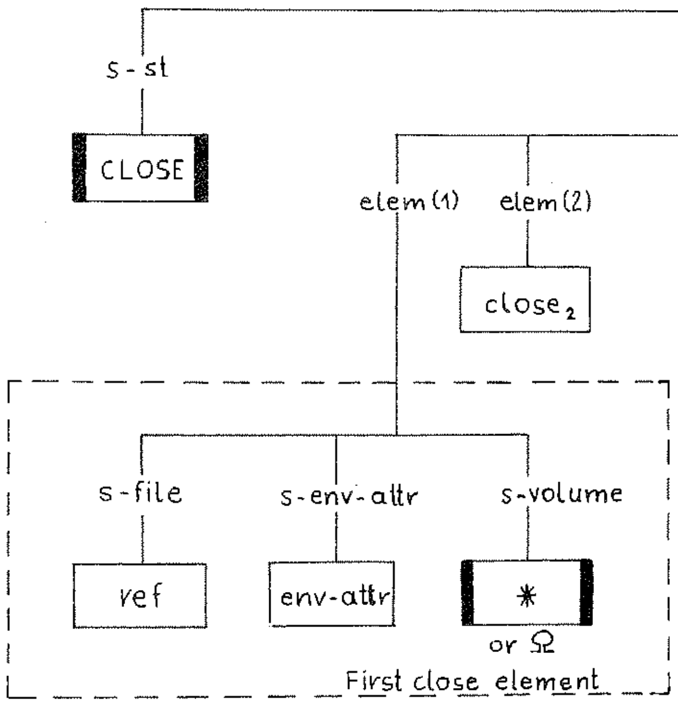


Fig. 12.2 Open and close statements

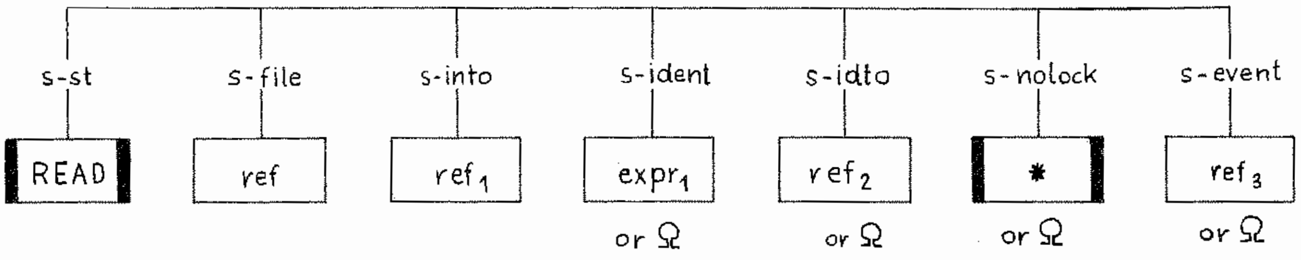
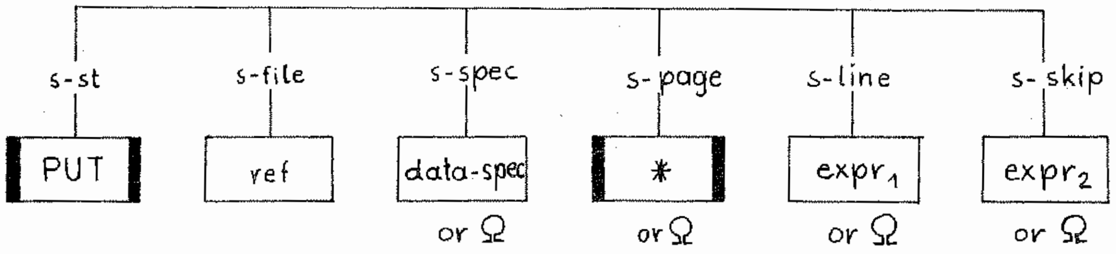


Fig. 12.3 Into-read statement



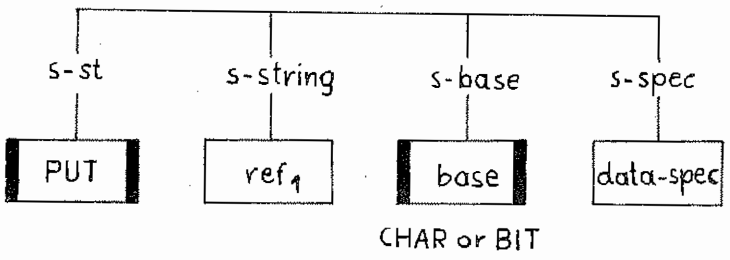


Fig. 12.4 File-put and string-put statement

It can be seen that I/O statement text is nearly a one-one translation from concrete text. The major exceptions are:

(1) In an <u>open element</u> the component s-lsz will optionally indicate the linesize. The distinction between LINESIZE and BLINESIZE can be taken from s-blsz (Ω or *).

(2) The component s-open-attr will always be a set of file attributes in the case of an open element (including the empty set). In this respect an open element always differs from a <u>close element</u>.

(3) The component s-env-attr originates from a specified concrete ENVIRONMENT option. It is presupposed that the translation will yield env-attr in some normalized but unevaluated form.

(4) In all I/O statement either a file reference, s-file, or a string reference (or expression in the case of string-get statement), s-string, is available. In the string case, the component s-base

allows the separation of a concrete STRING option from a BITSTRING option.

(5) The translator is assumed to insert the constant one in the case there is a SKIP option without expression in the concrete text.

(6) Record I/O and display statements have any KEYTO or REPLY options available under s-idto, any KEY, KEYFROM or DISPLAY options are available under s-ident.

The structuring of data specifications is described in 12.6.

Interpretation of I/O statements starts with a check of the statement text for mutually incompatible options (e.g., in any open or close element a non-empty component s-volume is in conflict with all components except s-file and a component s-open-attr which must in this case be the empty set) and for the incorrectness of single options (e.g. ref_1 in Fig. 12.3 must refer to a connected aggregate). Thereafter, those options which are expressions or environment attributes are evaluated in arbitrary order but one after the other. In particular, also options belonging to different open or close elements will be evaluated in arbitrary order. Evaluated options are integer values (for s-lsz, s-psz, s-ignore, s-skip, s-line), lists of character values (for s-title, s-ident), evaluated environment attributes (for s-env-attr), scalar event generations (for s-event), scalar character string generations or pseudo generations (for s-idto), generations (for s-from, s-into), file operands (for s-file in an open or close element), and file union names (for s-file in I/O statements other than open or close).

The evaluated options are inserted in the original statement text. For any I/O statement except open and close, the resulting object is called the <u>evaluated statement text</u> et. The file union name inserted as the s-file component is the result of normal implicit opening (cf. 12.2.2). The et is particularly helpful in the interpretation of record I/O statements.

<u>12.2 OPENING</u>

Corresponding sections of /5/:

3.6.1 The file directory <u>FD</u>

3.6.2 The file union directory <u>FU</u>

11.3.1 Open and close statement

11.3.2 Implicit opening

11.3.3 Opening

The following abbreviations are used in this section:

fa	set of file attributes
ea	evaluated environment attribute

id	identifier
st-prt,ST-PRT	standard print
tmt	transmit

Ordinary I/O statements can be properly interpreted only if a file has been opened which matches with the I/O statement. The classification of opening given in section 12.2.2 should relate the interpretation of ordinary I/O statements and open statements (started in 12.1) with the proper opening actions described in section 12.2.3. The most interesting case of proper opening is the creation of a file which causes changes in the file directory of the current task and in the file union directory. The function and structuring of both directories is anticipated in section 12.2.1 in order not to burden the description of proper opening.

12.2.1 FILE AND FILE UNION DIRECTORIES

The file directory FD serves two purposes:

(1) It links any file name with its evaluated environment attribute fd-ea, its file constant identifier id, and its file attribute fa (Fig. 12.5).

(2) It may link any file name (optional fd-status in Fig. 12.5) with an entry in the file union directory.

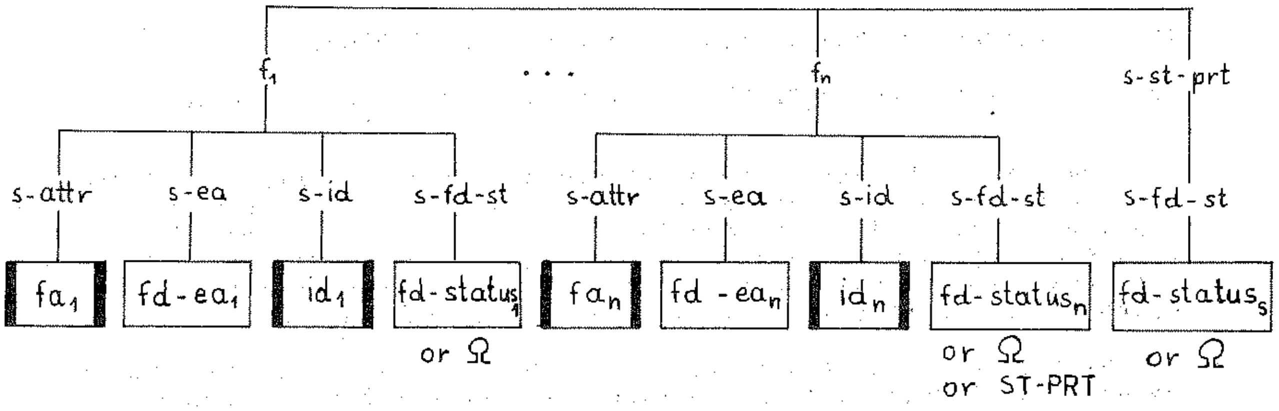


Fig. 12.5 File directory

The components fd-ea, id, fa are constant, and have been entered into the FD by the prepass. They are needed only as arguments for proper opening.[1] Successful proper opening amends the FD-entry under consideration by the fd-status (Fig. 12.6) which contains a file union name u, the indication that the file has been opened in the current task (*), and a component dealing with errors in stream data transmission. The first two components of the fd-status remain constant until the file closed. Closing deletes the fd-status.

---

1) fa is simply a copy of the file attributes available through application of an appropriate file value to the attribute directory.

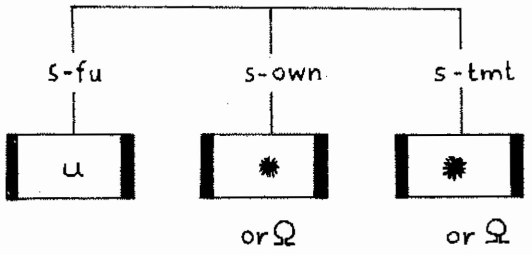


Fig. 12.6 Status of a file directory entry

Up to now only "program file names" $f_1,\dots,f_n$ have been discussed. There is one additional file name, the <u>standard system print file name</u> s-st-prt, which does not have explicit fd-ea, fa, and id components since all these components are constant for an implementation (fd-ea), and fa and id are constant for the language. Opening with the file name s-st-prt creates an fd-status, closing deletes it as usual.

The file name s-st-prt, as opposed to program file names, cannot be the denotation of any file value. Hence, the file name s-st-prt is not accessed <u>directly</u> as a result of an evaluated file option but only in the following cases:

(1) copy action because of copy option on file-get statement,

(2) standard system action for check on-condition,

(3) indirect access by the particular program file name f_n.[1]

Case (3) is the only case which necessitates some kind of linkage between the file directory entry for f_n and the entry for s-st-prt. This linkage is provided by the special fd-status ST-PRT (entered under the file name f_n), and the ordinary fd-status entered under the file name s-st-prt. Notice that f_n may refer <u>indirectly</u> to s-st-prt, but s-st-prt cannot refer indirectly to f_n as long as no linkage exists.

The file union name u is unique for a particular opening. Opening enters and closing deletes the file union selected by u in the file union directory <u>FU</u> (Fig. 12.7). File unions are described in section 4.3.2 (Figs. 4.10 and 4.11).

----

1) In Fig. 12.5 id_n corresponds to SYSPRINT and fa_n is one of the set of file attributes {STR}, {OUT}, {STR,OUT}, or {STR,OUT,PRT}.

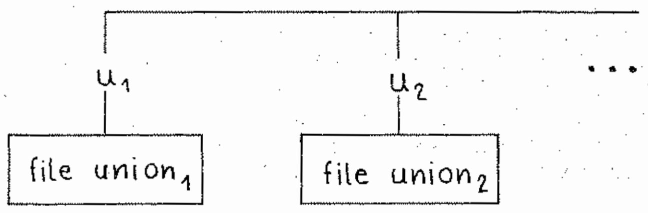


Fig. 12.7 File union directory

An attached task will be supplied with a copy of all components of the file directory of the attaching task which are not strictly private to a task. The strictly private components are s-own and s-tmt. This allows a classification of any fd-status into an <u>own</u> or <u>inherited fd-status</u>. As a consequence of copying the file union names, file unions might be shared by tasks.

<u>Example</u>:

```
MAIN:PROC ...;
     PUT DATA;
     CALL P TASK;
     CLOSE FILE(SYSPRINT);
   P:PROC;
     GET LIST(X) COPY;
     END MAIN;
```

The PUT and CLOSE statements refer indirectly to the standard system print file name f_n. Hence, the file is opened by the PUT statement, will be inherited to the task P, and is closed by the CLOSE statement in the attaching task MAIN. The COPY option of the GET statement will not create a new file union. Either it will refer directly to the standard system print file (if it is still open), or the interpretation of the COPY option will be erroneous if the standard system print file has been already closed.

If the example is modified, and the PUT statement is executed in the attaching task <u>after</u> the call of task P, then the PUT statement and the GET statement will create two independent file unions.

12.2.2 TYPES OF OPENING

There are three types of opening:

(1) explicit opening caused by a single open element,

(2) implicit opening caused by ordinary I/O statements,

(3) implicit and direct opening of standard system print files.

Explicit opening by a particular open element may take place if all options of the open element have been evaluated, and if explicit opening of any open elements to the left has been completed. After completion of explicit opening the open element will be deleted from the statement text or the arguments for transmit or undefinedfile on-condition calls will be inserted in place of the open element. If opening of all open elements is completed then the modified statement text will be inspected from left to right for on-condition arguments, and the on-conditions will be called.

Explicit opening supplies proper opening with the following arguments:

(1) the file value n (taken from s-file),[1]

(2) the set of attributes derived from s-open-attr and the set of file attributes declared, i.e., contained in the FD-entry for the file name n(DN), shortly f in the sequel,

(3) the data set title s-title (if non-empty), otherwise the file constant identifier contained in f(FD),

(4) the evaluated environment attribute merged from s-env-attr and the evaluated (declared) environment attribute in f(FD),[2]

(5) the volume option s-volume,

(6) the line and page sizes: s-lsz, s-blsz, s-psz.

Implicit opening may take place if the file option of the ordinary I/O statement has been evaluated, and if evaluation yielded a file operand. Implicit opening may yield an open file or it may immediately cause transmit or undefinedfile on-condition calls. After returning from an undefinedfile on-condition call, the file might have been opened. In all cases where an open file is left, a final check will be made as to whether the file is consistent with all statement options, and the file union name will be returned.[3]

Implicit opening supplies proper opening with the following arguments:

(1) the file value n taken from the file operand,

(2) the set of attributes derived from the attributes deduced from the statement and from the file attributes contained in f(FD),

(3) the data set title identical with the file constant identifier contained in f(FD),

(4) the evaluated environment attribute identical with that contained in f(FD),

---

1) The mention of "s-file", etc. means the "component s-file of the pre-evaluated text", etc.
2) s-env-attr might contribute to the evaluated environment of the file union in a similar way as s-open-attr does to the complete set of attributes.
3) The check is always necessary since the deduced file attributes depend on the statement type (s-st) but do not depend on the statement options.

(5) the volume option does not apply (IMPL),

(6) the line and page sizes do not apply (Ω).

Direct and consequently <u>implicit opening of a standard system print file</u> supplies the following arguments to proper opening:

(1) the file value does not apply (Ω),

(2) the set of attributes is {CST,OUT,PRT},

(3) the data set title corresponds to SYSPRINT,

(4) the evaluated environment attribute is implementation-dependent,

(5) and (6) are the same as for implicit opening.

12.2.3 PROPER OPENING

<u>Proper opening</u> first makes an error test on its arguments and creates a unique name u. The second step tests whether a new file is to be opened, and if so makes all entries in <u>FD</u>, <u>FU</u>, and external storage <u>ES</u> (cf. 12.2.3.2). The second step may be unable to open a new file either because a file is already open or because the opening criterion is violated. The third step returns the arguments for undefinedfile on-condition calls if the second step did violate the opening criterion. In all other cases the actions of the third step will depend on the volume option (Ω, *, IMPL) and the <u>status</u> of the file union. These actions are described in section 12.4.

<u>12.2.3.1 Opening criterion</u>

The <u>opening criterion</u>, i.e., requirements (1) through (4) and optionally (5) must be satisfied if successful opening should occur:

(1) The set of attributes, being an argument of proper opening, is a <u>complete set of attributes</u>. The complete sets of attributes can be taken from the description of the mapping parameter in 4.3.2. However, in all instances where the attribute SEQ or TRA is a member of an attribute set, one of the buffering attributes BUF or UNB has to be added. In addition, {REC,DIR,KEY,UPD,EXC} is a complete set of attributes.

(2) The data set title and the (merged) evaluated environment attribute, being arguments of proper opening, access a data set ds in external storage <u>ES</u> (cf. 4.3.1).

(3) There exists a mapping [1] dependent on ds and the mapping parameter mp, where mp is composed of the (merged) evaluated environmemt attribute, the data set title, and the complete set of attributes properly adjusted.

(4) There exists a mapping dependent on the data set ds_1 and mp, where ds_1 has a mapping number which is one greater than the mapping number of ds (both mapping numbers with respect to mp).

---

1) cf. footnote in 4.3.3.1.

(5) Indirect opening of the standard system print file is only successful if the standard system print file is not open.

12.2.3.2 Successful proper opening

The effect of successful proper opening with a new file union name u and some file name f is summarized below:

(1) The fd-status is entered under f in FD as described in 12.2.1. In case of indirect opening of the standard system print file two entries are made.

(2) The file union is entered under u in FU. The components s-p (file parameter) and s-f (file name) [1] have been described in 4.3.2. Of all additional components a file union may have (cf. the compilation in Fig. 4.11), only the status (st), the current column (col), the linesize (lsz), the current line (line), the pagesize (psz), or the names of attached I/O-events (io-ev) are part of the initial file union.

The components col and line are initially one, io-ev is initiated with the empty set, lsz and psz are set according to the arguments of proper opening, st is set to SW-BOV (cf. 12.2.3.3).

An important property of the file union (throughout its entire existence) is the compatibility of all its components with the complete set of attributes contained in the file parameter component. Hence, for example, col might be an integer in case the attributes BST or CST are specified otherwise col is empty; the component tn-key (cf. Fig. 4.11) is a directory of keys only if EXC is specified, etc.

(3) The data set ds accessed is replaced by the data set ds_1 described in 12.2.3.1(2,4).

12.2.3.3 File union status

The file union status st (not to be confused with the fd-status) characterizes the transition of the file union with respect to data set label processing and data set switching. Fig. 12.8 shows the possible values of st and how they may be reached. Reading and writing of data set labels, basic data transmitting actions (cf. Fig. 4.17), and data set switching is checked if st actually conforms with the particular action to be performed. This eliminates errors in case the file union is shared over tasks.[2]

---

1) This copy of the file name in the file union is a convenience and not a necessity.
2) For example, multiple processing of a label of one and the same data set is excluded.

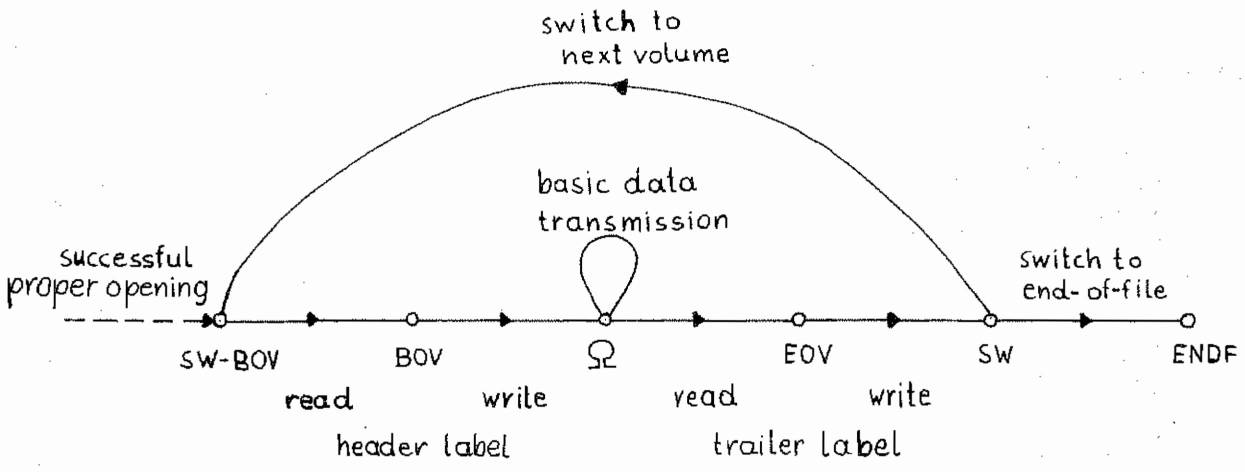


Fig. 12.8 File union status

12.3 CLOSING

This section is arranged similar to the section on opening. In all instances where closing is similar to opening except for some straightforward changes, a description will be omitted.

12.3.1 TYPES OF CLOSING

There are two types of closing:

(1) explicit closing caused by a single close element,

(2) implicit closing by the epilogue of that task which opened the file.

Explicit closing is like explicit opening, except that no undefinedfile on-condition call can result. Proper closing is provided with the following arguments:

(1) the file name f (taken from the component s-file of the pre-evaluated text and DN),

(2) the evaluated environment attribute merged from the component s-env-attr of the pre-evaluated text and the evaluated environment attribute of the file union,

(3) the volume option s-volume of the pre-evaluated text.

Implicit closing depends exclusively on the FD of the task whose epilogue is in progress. Proper closing is provided with the following arguments:

(1) a file name f having an own fd-status,

(2) an evaluated environment attribute which is implementation-dependent,

(3) the volume option does not apply (IMPL).

12.3.2 PROPER CLOSING

Proper closing first makes a case distinction which separates successful proper closing in case of no volume option (Ω or IMPL) and an own fd-status from no action (if the fd-status is empty), and trailer label processing followed by data set switching if there is a volume option (*) which is compatible with the file (cf. 12.4, condition type EOV).

Successful proper closing can be separated into three steps. The first step transmits the buffer and/or frees the buffer registered in the file union of a buffered file, and deletes any still active I/O-events registered in the file union. Buffer transmission is similar to the first part of the execution of an evaluated locate statement. Hence, interpretation of locate transmission will allow for an artificial CLOSE-LOCATE statement (cf. 12.5.3.2).

The second step is concerned with trailer label processing and data set switching. This step is skipped if the status of the file union is not empty (cf. Fig. 12.8) or if the step is part of a task epilogue.

The third step is the reverse process of successful proper opening: The fd-status and the file union are deleted, and the data set accessed is replaced by a data set whose mapping number is one smaller than before. In case the standard system print file is closed, and if it had been opened indirectly, of course the special fd-status ST-PRT will be deleted, too.

12.4 LABEL PROCESSING AND DATA SET SWITCHING

Data set label processing with or without data set switching depends on a file union name and a condition type which is BOV, EOV or EOV-BOV.

The condition type BOV indicates that a header label is to be read, passed to a begin of volume on-condition call, and is to be written upon leaving the on-unit. This corresponds to the status transition SW-BOV, BOV, Ω in Fig. 12.8. These actions are performed as third step of proper opening if the volume option is not empty and the status is SW-BOV. After reading the label in case of sequential input or after writing the label in case of sequential update, the data set is positioned to position zero.

The condition type EOV indicates that a trailer label is to be read, is passed to an end of volume on-condition call, is to be written upon leaving the on-unit, and the data set is to be switched. This corresponds to the status transition Ω, EOV, SW, SW-BOV or ENDF. Data set switching transforms the status from SW to ENDF if the file union specifies the attribute KEY or if it specifies INP or UPD and the accessed data set is the last volume. The checking as to whether a data set is the last volume is implementation-defined, and depends on the mapping parameter, the data set, and the current volume number volno of the file union (cf. Fig. 4.11). Data set switching transforms the status from SW to SW-BOV in all other cases, and increments volno appropriately.

Notice that data set switching causes no change of the data set. Hence, it would be more precise to speak of file union switching. The data set might be changed by environmental influences.

The conditon type EOV-BOV indicates that the actions corresponding to EOV and BOV should occur in succession. The BOV actions are cancelled if the EOV actions have not reached the status SW-BOV. The EOV-BOV actions are performed

(1) as third step of proper opening if the volume option is * and the status is other than SW-BOV (most reasonably Ω),

(2) as part of record transmission or stream transmission if the end of the data set has been reached by a previous basic data transmitting action.

12.5 RECORD TRANSMISSION

Corresponding sections of /5/:

11.4.2 Diaplay and record handling statements

11.5 Record transmission

The following abbreviations are used in this section:

n	name
en	event name
tn	task name
o	pointer or offset
ap	area pointer
ptr	pointer reference
mp	mapping parameter
ds	data set
el	proper data element
cbif	on-condition built-in function

Record transmission depends on the evaluated statement text et (cf. 12.1) and the list of those references ref-list for which check on-conditions are to be raised in the sequel but which are not contained anymore in et. In particular, s-file (et) is the file union name.

The following case distinctions are made:

(1) If the file union status is ENDF then the endfile on-condition will be called.

(2) If s-event(et) is an event generation then an I/O-event will be attached. This depends on et, ref-list, and a newly created event name en (cf. 12.5.1).

(3) If et is an evaluated unlock statement then the specified key, s-ident(et), will be unlocked immediately (cf. 12.5.2).

(4) Otherwise the following actions will occur in succession:

(a) Proper data transmission depending on et, and the returning of a list of condition indications (cf. 12.5.3.1). Condition indications are arguments to on-condition calls or they contain the special indication END. They indicate unusual situations and/or transmission errors.

(b) Call of the on-conditions for which indications have been returned by step (a).

(c1) If the special indication END has not been returned by step (a): Conditional unlocking of key and check on-condition calls for ref-list. This terminates interpretation of the statement.

(c2) Otherwise: Step (4) is retried. This is preceded by a call of the pending on-condition and a wait for further input (in case the attributes TRA and INP are contained in the file union) or by data set label processing and data set switching of condition type EOV-BOV (in all other cases). Data set switching and further input usually depend on environmental influences. Hence, it will depend on these whether step (c1) will ultimately be taken.

12.5.1 I/O-EVENTS

An I/O-event is a special kind of parallel action which is able to execute proper data transmission "in parallel" with other actions (cf. chapter 7).

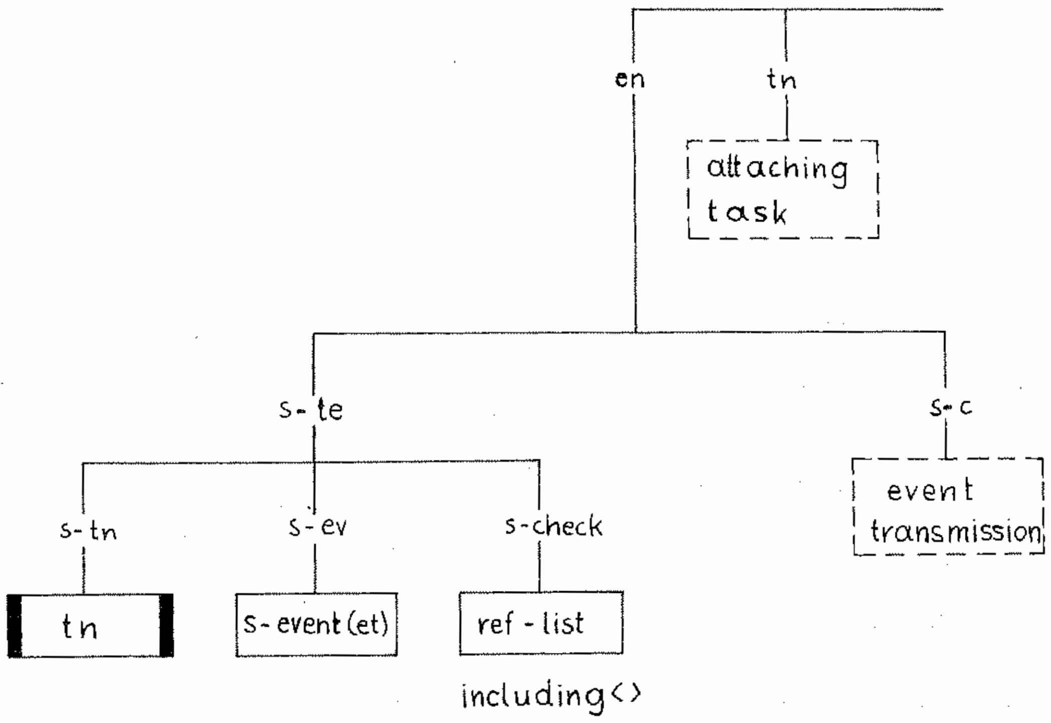


Fig. 12.9 Part of PA showing an I/O-event just created

Fig. 12.9 shows the entry made in PA at attaching [1] an I/O-event characterized by the event name en, the evaluated statement text et, and ref-list. Besides that, attaching of an I/O-event causes an assignment to the event generation, and the addition of en to the component io-ev of the file union (cf. Fig. 4.11, and 12.3.2) and to the relevant component of TE.

The name of the attaching task (component s-tn) is only used in connection with locking of keys.

The box "event-transmission" in Fig. 12.9 denotes the actions of proper data transmission depending on et followed by conditional unlocking and some terminating actions. These terminating actions

(1) handle the returned list of condition indications, and insert the list of arguments to on-condition calls under s-cond, and also insert et under s-eov-bov if the special indication END has been returned [2] (cf. Fig. 12.10);

---

1) Creating, activating, starting are used as synonyms of attaching.
2) This may occur for write, into-read, or ignore-read statements.

(2) activate all parallel actions, since the I/O-event has performed all its actions (the component s-c will be empty), and any other parallel actions which might have been waiting for the I/O-event to become semi-complete could continue.

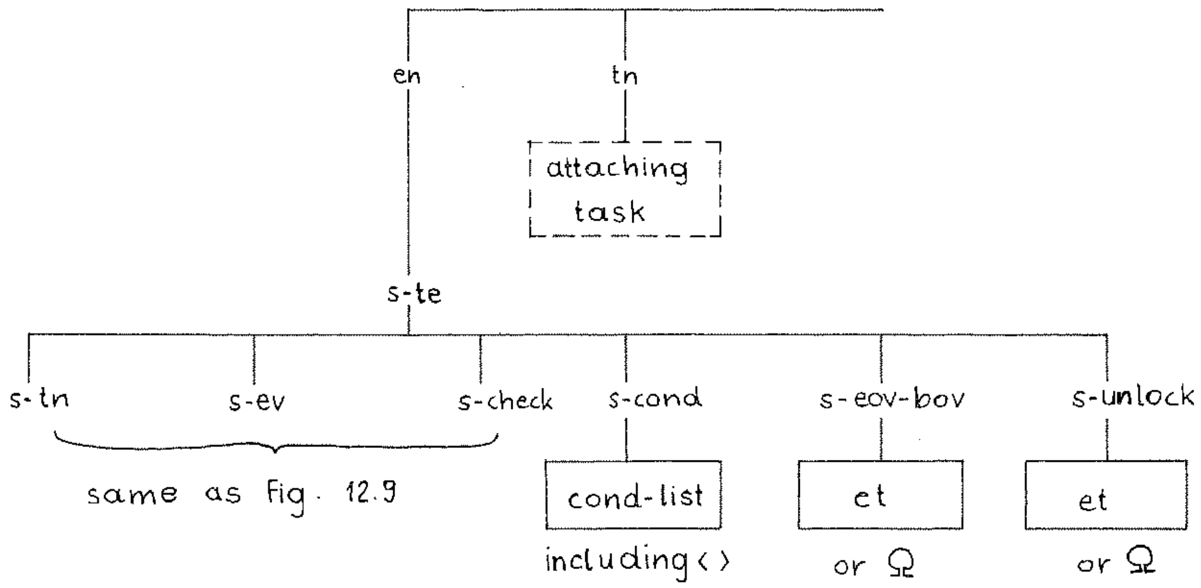


Fig. 12.10 Part of PA showing a semi-complete I/O-event

Completion of the semi-complete I/O-event by a wait statement (cf. Fig. 7.7) executes I/O on-condition calls (component s-cond), performs immediate unlocking (component s-unlock), sets the completion value of the event variable associated with the I/O-event to "complete" and deletes the entry for the I/O-event from PA (analogous to 7.4, (6,7)), retries the data transmission but without attaching a new I/O-event (essentially the same actions as step (4), (c2) of the indroduction to 12.5), and executes check on-condition calls (component s-check).

12.5.2 LOCKING OF KEYS

Any file union containing the attribute EXC may have a directory the entries of which are sets of keys. The entries are selected by task names. A particular key (i.e., list of character values) is locked by task tn if it is a member of a directory entry selected by tn. In particular, the key is locked-own if tn is the name of the current task TN or, in case the question is posed during the interpretation of an I/O-event, tn is the name of the attaching task s-tn(TE). The key is locked-foreign if it is locked by some task but not locked-own.

The first step of proper data transmission, which is applicable to EXC files only, checks if the key, s-ident(et), is locked-foreign. If so, a wait takes place until the key will be unlocked by the task for which the key is locked-own (immediate or conditional unlocking, see below). Otherwise, or after the wait, the key will be entered in the directory, and will be locked-own from then on. However, this entry is made only if the statement at issue has no nolock option, i.e., s-nolock(et) is empty.

<u>Conditional unlocking</u> has no effect except if it is performed by a delete, rewrite, or write statement on an EXC file. If the statement has no event option then immediate unlocking will be performed, otherwise et will be inserted under s-unlock of the event specification <u>TE</u> (cf. Fig. 12.10).

<u>Immediate unlocking</u> has no effect except if it is performed on an EXC file, and the key is locked-own. The key will be deleted from the directory in the file union, and all waiting parallel actions are activated.

The deletion of the whole file union which occurs at successful proper closing, and the deletion of all keys locked by some task which takes places at the termination of that task (cf. 7.4,(3)) are special cases of immediate unlocking.

12.5.3 PROPER DATA TRANSMISSION

The actions designated by <u>proper data transmission</u> comprise all activities which have to do with the transmission of a particular record data element between internal and external storage. These activities include a transition of the data set causing a modification of <u>ES</u>, freeing and/or allocation of a buffer and/or several assignments causing a modification of <u>S</u>, a modification of the file union component dealing with buffers, and the construction of a list of condition indications.

Proper data transmission depends on et. It is defined for write statements (Fig. 12.11), locate statements including the artifical CLOSE-LOCATE statement (Fig. 12.14), rewrite, read (i.e., set-read, into-read, ignore-read), and delete statements (Fig. 12.15).

The organization of this section follows the flow outlined in the figures. Actions denoted in the flow charts by names ending on "-transmission" refer to the basic data transmitting actions mentioned in section 4.3.4 (Fig. 4.17). Such actions may be performed only if the file union status is empty (Fig. 12.8).

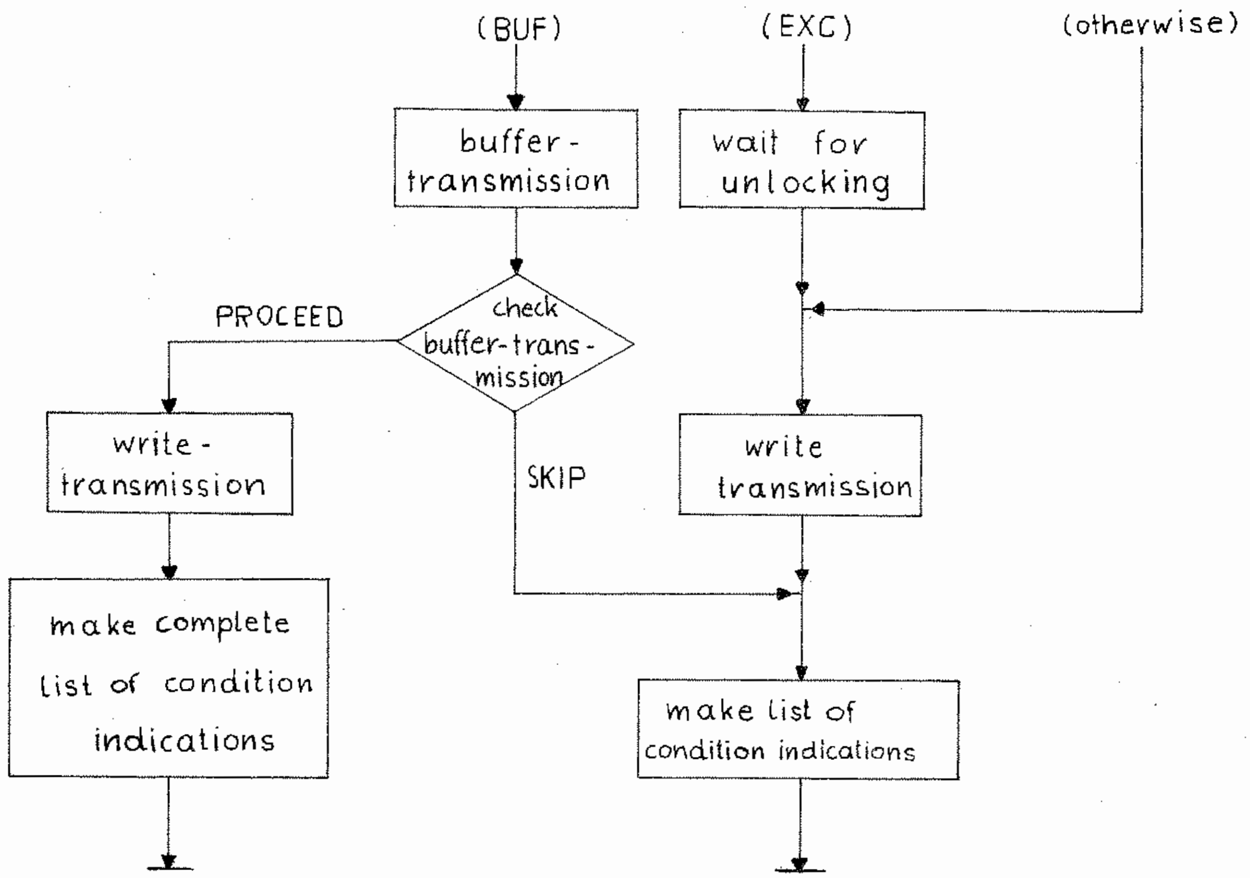


Fig. 12.11 Proper data transmission - WRITE

12.5.3.1 Write

The actions differ as to whether the file union contains the attribute BUF, EXC, or none of both. The boxes of Fig. 12.11 have the following meaning:

(1) Wait for unlocking. This potential wait is described in 12.5.2 as first step of proper data transmission.

(2) Write-transmission. The basic data tranmitting function write (cf. 4.3.4.3) applied to its arguments mp, ds, el yields the new data set and possibly an indication for one of the unusual situations KEY, REC, END. The argument mp is the mapping parameter, ds is the (old) data set, and el is the record data element having (a possibly empty) key component, s-ident(et), and a value representation component holding the storage designated by the generation s-from(et).

The new data set replaces ds in ES, and a set containing an indication for an unusual situation ¹ together with any transmission error flag (cf. 4.3.4.4) is passed to the following step.

(3) <u>Make list of condition indications</u>. The set of indications is modified and ordered. The resulting list contains elements which serve as arguments to on-condition calls or they contain the special indication END (Fig. 12.12). The file name f is taken from the file union, io-cond is one of the elementary objects KEY, REC, TMT, and END, and the component s-cbif provides those values to condition built-in functions which are characteristic for the situation.

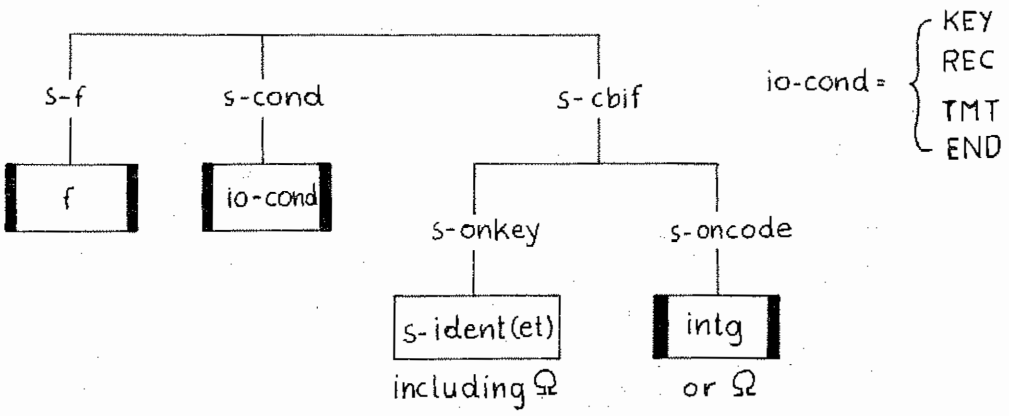


Fig. 12.12 Condition indication made by WRITE

(4) <u>Buffer-transmission</u>. The basic data transmitting function write will be used analogously to (2) but the record data element el is a <u>buffer</u>.

Any file union containing the attribute BUF may contain a component (buf in Fig. 4.11) which specifies one or two pointers, and a key consisting of a list of characters (only if the attributes KEY and OUT are also specified). If only one pointer o (Fig. 12.13) is present in the file union then it denotes the main storage o(<u>S</u>). Otherwise the area storage o•ap(<u>S</u>) is designated where ap is the area pointer and o is the offset.

---

1) Instead of "REC" an integer value is passed which results form the comparison of the record- and storage-sizes involved.

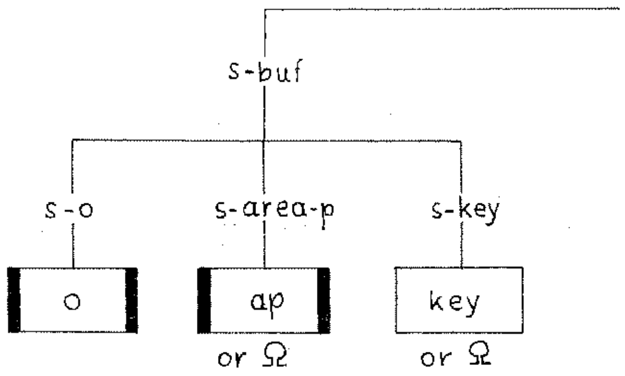


Fig. 12.13 File union entry for an allocated buffer

Buffer-transmission is skipped (i.e., no action) if the file unions buffer component is empty. In all other cases the data element el may be constructed. If the function write does not yield one of the unusual situations KEY or END then the new data set replaces the old one in <u>ES</u>, and a set of condition indications is passed to the following step just as detailed in step (2). In addition the <u>buffer</u> is <u>freed</u> and deleted from the file union.

If the function write yields one of the unusual situations KEY or END the same actions are performed except that the buffer is neither freed nor deleted.

(5) <u>Check buffer-transmission</u>. The exit labelled SKIP is taken if unusual situations KEY or END occurred in the previous buffer-transmission, otherwise the exit PROCEED is taken.

(6) <u>Make complete list of condition indications</u>. The two sets of condition indications resulting from buffer- and write-transmission are combined and are treated in a similar way as described in step (3).

12.5.3.2 Locate

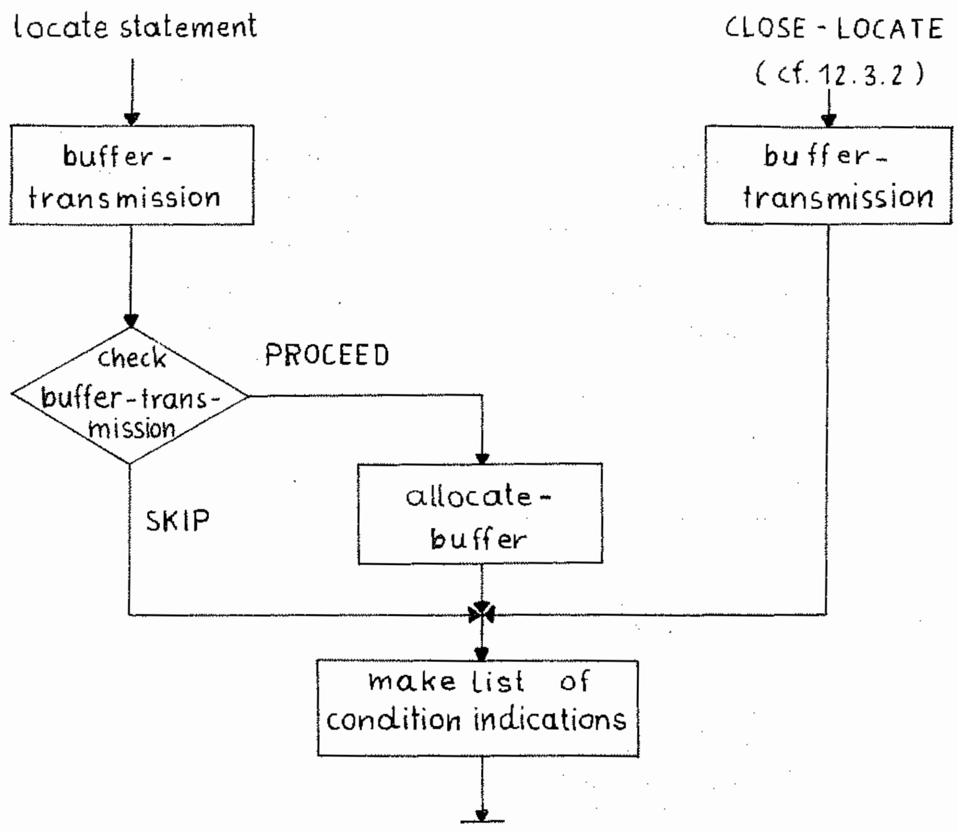

locate statement
CLOSE-LOCATE (cf. 12.3.2)
buffer-transmission
buffer-transmission
check buffer-transmission
PROCEED
SKIP
allocate-buffer
make list of condition indications

Fig. 12.14 Proper data transmission - LOCATE

There is only one action namely the <u>allocation of a buffer</u> (box allocate-buffer in Fig. 12.14) which has not yet been described in section 12.5.3.1.

Allocation of the based varible with unique name s-n(et) is either in main storage or in an area dependent on s-ptr(et), or in absence of a set option, on the declaration s-n(et)(<u>AT</u>) of the based variable. The components s-id, s-n, and s-ptr [1] of et and these same components of a specification of a single based allocation have analogous functions. Hence, the corresponding description of actions can be taken over from chapter 10 if the following differences are observed:

In case allocation is in main storage the type of allocation is BUFFER (instead of BASED), and the buffer pointer and/or key is entered into the file union (o, key in Fig. 12.13) instead of being added to the based free set of <u>TE</u>.

---

1) s-ptr(et) has not been evaluated previously. It is still a reference or empty.

In case allocation is in an area the area pointer, the offset and/or key are entered into the file union (ap, o, key in Fig. 12.13).

12.5.3.3 Rewrite

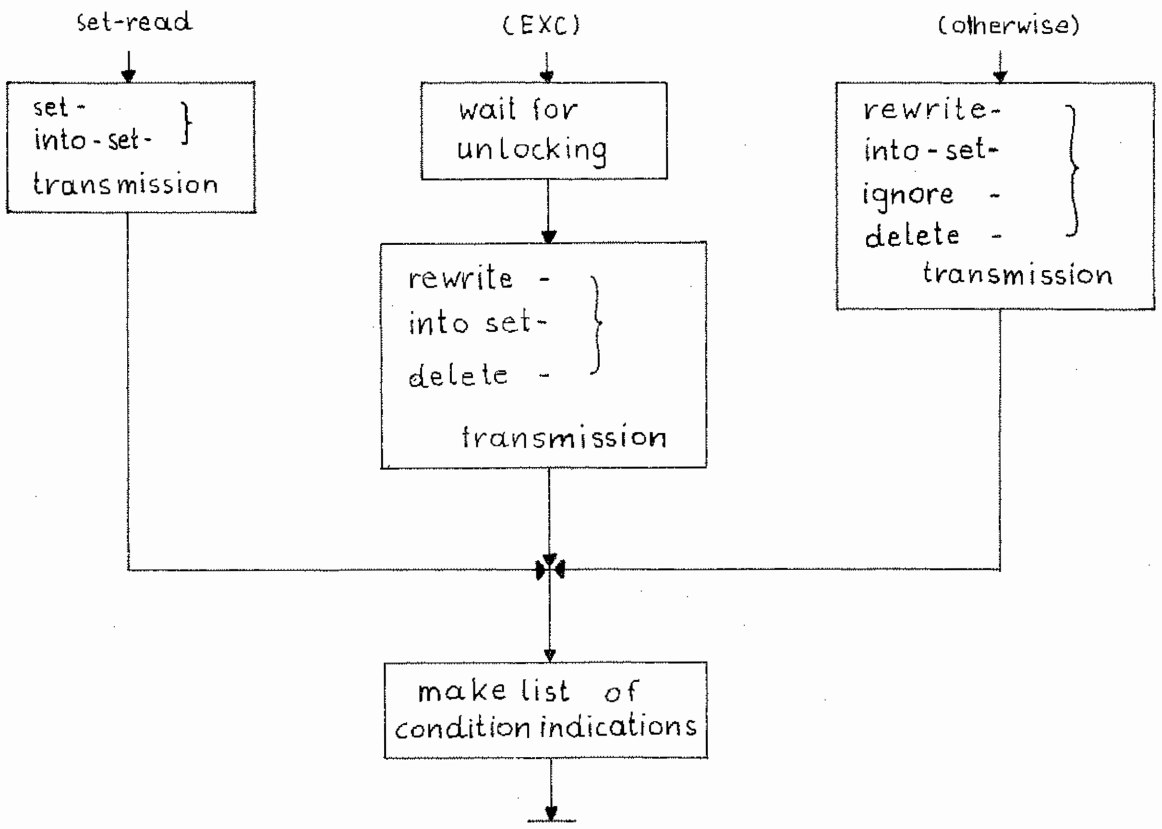

set-read
(EXC)
(otherwise)
set-into-set-transmission
wait for unlocking
rewrite-into-set-ignore-delete-transmission
rewrite-into set-delete-transmission
make list of condition indications

Fig. 12.15 Proper data transmission - REWRITE, READ, DELETE

The actions correspond to the flow chart of Fig. 12.15, the middle branch being taken if the file union contains the attribute EXC, the right branch is taken in all other cases.

Wait for unlocking and the making of condition indications has been dealt with in 12.5.3.1 (1) and (3).

The <u>rewrite-transmission</u> uses the basic data transmitting function rewrite (cf. 4.3.4.2). The arguments are mp, ds (as described in 12.5.3.1), and a record data element el which is built from the storage designated by s-from(et) or from the buffer. The condition indications KEY or REC may result. The replacement of the old data set by the new data set in <u>ES</u>, and the handling of on-conditions is analogous to 12.5.3.1.

12.5.3.4 Set-read

The actions for a set-read statement and for a locate statement have some similarities: Both statements are restricted to file unions containing the attribute BUF, and both statements comprise data transmission, buffer freeing, buffer allocation (in main storage or in an area), and the insertion of the new buffer in the file union.

Since loacte is restricted to OUT and set-read to INP or UPD, the statements may not be related with one and the same file union. In case of locate the size of the buffer to be allocated depends on the evaluated aggregate attribute of the based variable. However, for a set-read the size is derived from the record data element read.

The case distinction as to whether allocation for a set-read is to occur in main storage or in an area is made on the basis of the set option s-ptr(et). In the first case the treatment is very similar to an into-read on a buffered file (cf. 12.5.3.5, into-set-transmission and left branch of Fig. 12.15). In the latter case the record data element is read, i.e., the basic data transmitting function read (cf. 4.3.4.1) is applied to mp, ds, and s-ident(et). This yields the new data set, and an indication for one of the unusual situations KEY, END or (this is the usual situation) the data element read in.

(1) Unusual situation: The new data set replaces the old one, and indications are passed to the following step.

(2) Usual situation: In addition to (1) the old buffer (if any) is freed, a new buffer, with allocation type AREA, is allocated (if possible), the value representation component of the data element read in is assigned to the buffer and the key component is assigned to the keyto option s-idto(et) (if applicable). The offset is assigned to the set option s-ptr(et), and the area pointer and the offset are entered into the file union (ap, o in Fig. 12.13).

12.5.3.5 Into-read, ignore-read, delete

The actions correspond to those described in 12.5.3.3 except that <u>into-set-</u>, <u>ignore-</u>, and <u>delete-transmission</u> is performed in place of rewrite-transmisssion, i.e., the basic data transmitting functions read, ignore, and delete are used, respectively (cf. Fig. 12.15).

No general description of the transmission actions is given since it follows in a rather straightforward way from a re-interpretation of the above sections and from the relevant sections of 4.3.4.

It should be noted that ignore- and delete-transmission have no effect on buffers registered in the file union; assignment to any keyto option s-idto(et) as part of into-set-transmission occurs only if the assignment and conversion rules yield no on-condition calls.

12.6 STREAM TRANSMISSION

Corresponding sections of /5/:

11.6 Stream transmission

11.7 Special cases of stream transmission

The following abbreviations are used in this section:

gen,ps-gen	generation or pseudo-generation
hi	higher index (current position)
lo	lower index
intg	integer value
fol	format list
init	initial
incr	increment
char	character value

12.6.1 INITIATION AND TERMINATION OF PUT AND GET STATEMENTS

Evaluation of statement options (cf. 12.1) affects the file, skip, and line options of put and get statements (Fig. 12.4). The resulting evaluated text et has a file union name u as its component s-file, and integer values as its components s-skip or s-line. It should be noted, that the data specification, s-spec(et), and all options of string-put or string-get statements are left unevaluated.

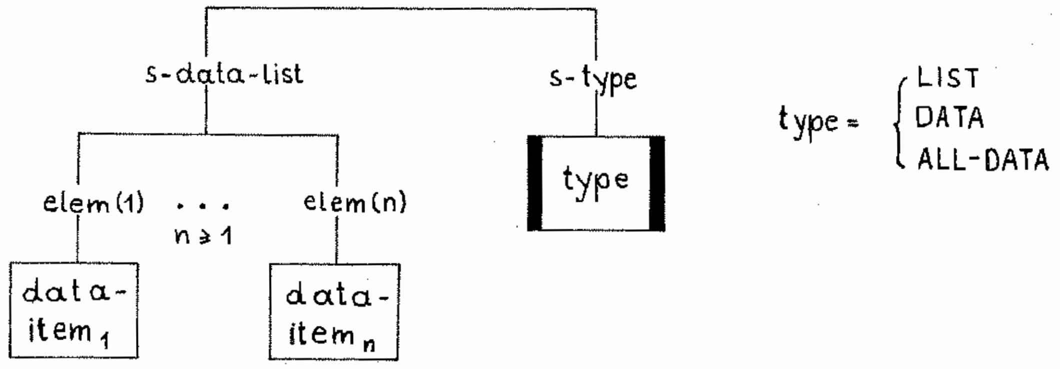


Fig. 12.16 List-or data-directed data specification

The structuring of list- or data-directed data specifications (Fig. 12.16), and edit-directed data specifications (Figs. 12.17 to 19) shows the close correspondence to the structuring of concrete text. There is only one non-trivial difference between concrete and abstract text which concerns data-directed data specifications with missing (concrete) data list: The translator is assumed to insert a data list containing all unsubscripted fully qualified references if they refer to proper variables which are not parameters and which are known in the block where the statement is executed; in such a case the type will be ALL-DATA. If there is a data list in the concrete data-directed data specification then the type will be DATA, and the data list is the one-one translation of concrete text.

The type ALL-DATA is needed in the interpretation of put statements in order to know that the ordering of the data list is irrelevant, and that operands which cannot be converted to character string are to be skipped.

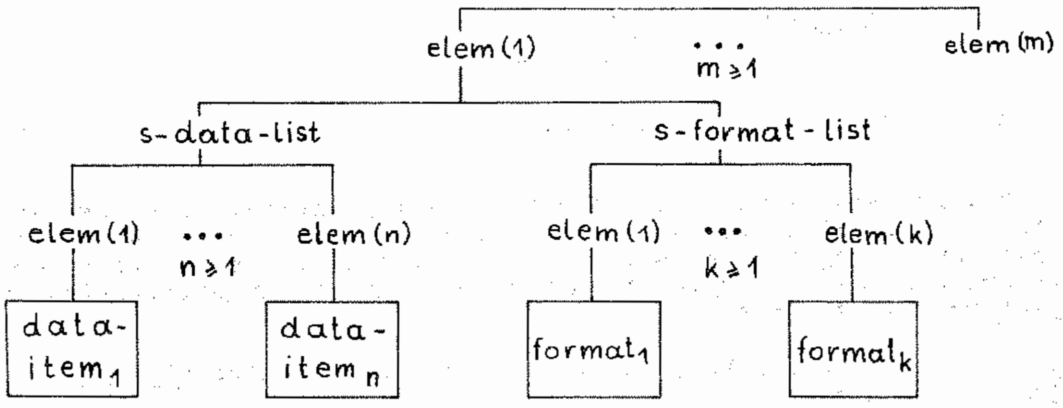


Fig. 12.17 Edit-directed data specification

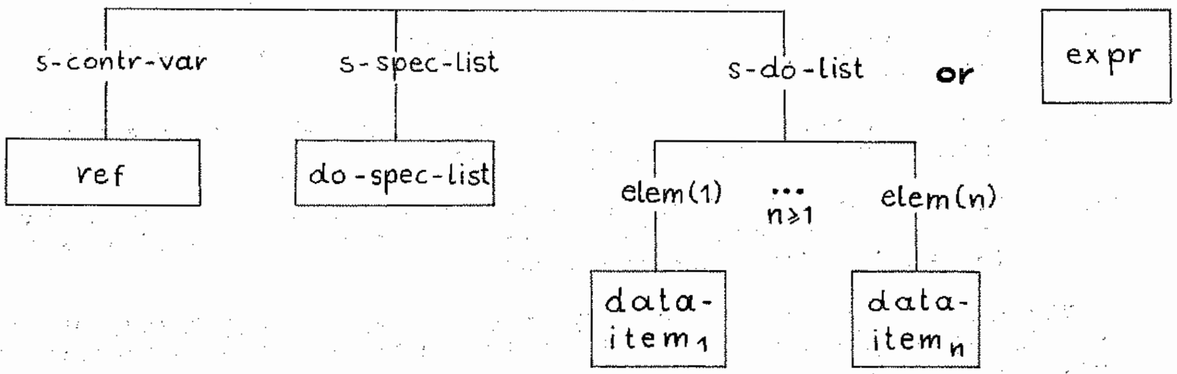


Fig. 12.18 Controlled or simple data items

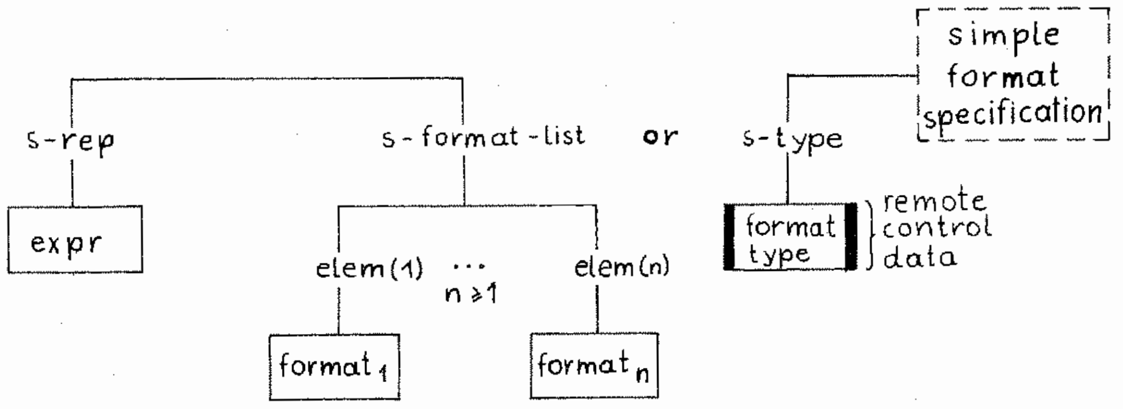


Fig. 12.19 Iterated or simple format

Initiation and termination of get and put statements depends largely on the presence of a string or a file option. However, interpretation of the data specification is only slightly affected by this difference ("expansion of data specification") except for the interpretation of elementary transmission.

The following initiating actions occur for evaluated <u>file-put</u> and <u>file-get</u> statements:

(1) If the file union status is ENDF then the endfile on-condition will be called. This terminates the statement.

(2) The component "count" of the file union is initiated to zero (cf. Fig. 4.11).

(3) Any page, skip, or line options are executed just like the corresponding simple control formats.

(4) The data specification is interpreted, i.e., expansion of the data specification occurs iteratively with transmission of data fields (except for data-directed input).

The initiating actions for <u>string-put</u> and <u>string-get</u> statements are preceded by a case distinction depending on the components s-string and s-base (cf. Fig. 12.4):

(a) The component s-string is a reference to a scalar string type variable or pseudo-variable, and the component s-base agrees with the type of the reference.

(b) A string-get for which (a) does not apply.

(c) An erroneous string-put.

If case (a) applies then the following steps will be taken:

(a1) The generation or pseudo-generation of the reference is evaluated, and it is entered under a newly created unique name into the file union directory <u>FU</u> (component s-g in Fig. 12.20).

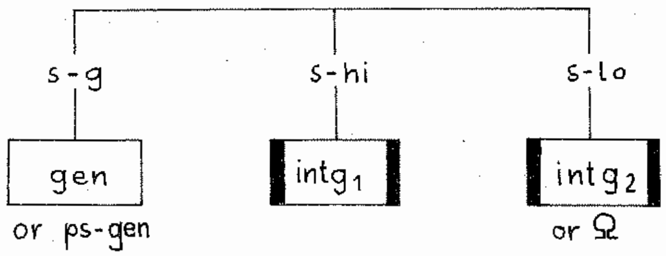


Fig. 12.20 "File union" corresponding to a string-put or string-get statement

This entry additionally contains a component s-hi of value zero. Transmission of data fields to or from the storage designated by the component s-g increments the component s-hi, i.e., the component has a similarity with the position of an inner data set (cf. 4.3.3.1). The component s-lo is an integer value in some cases of get statements: The component s-lo indicates the lowest, and s-hi indicates the highest position of the data field being read. The knowledge of both positions is necessary for the proper interpretation of onsource and onchar pseudo-variables and built-in functions used in conversion on-units called because of inconsistences in the data field.

(a2) Essentially the same actions as in step (4) above.

(a3) The check on-condition is called for the reference in case of a string-put.

If case (b) applies then the following steps will be taken:

(b1) The operand of the expression (expression includes reference) is evaluated and converted to a string with BIT or CHAR base as specified by s-base in Fig. 12.4.

(b2) A dummy is allocated (having the generation gen), and the converted operand is assigned to it.

(b3) Essentially the same actions as in step (a1) above with the generation gen.

(b4) Essentially the same actions as in step (4) above.

(b5) The dummy is freed.

Step (4), the interpretation of the data specification, is dealt with in the following sections. In these sections it will not be the complete data specification which is of major importance but the current data list or format list or parts of them (components s-data-list and s-format-list in Fig. 12.16 and Fig. 12.17). On the other hand, the data and format lists do not convey all the information necessary to characterize the data transmission. Hence, this information is collected in the transmission parameter (Fig. 12.21) which is a modified skeleton of a get or put statement: The components s-base, s-spec, s-page, s-line, and s-skip have been deleted from the statement text. The mutually exclusive components s-file and s-string contain the file union name relevant for the interpretation of the statement. An additional component s-type may be present in the transmission parameter which is a copy from the data specification (cf. Fig. 12.16) or is the elementary object EDIT in case of an edit-directed get or put statement.¹

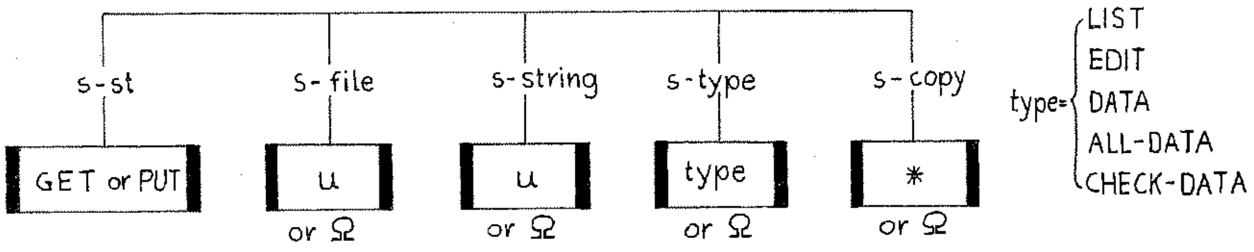


Fig. 12.21 Transmission parameter

12.6.2 DATA SPECIFICATIONS

An edit-directed data specification is a list which is interpreted from left to right. The first action is to put the format list of the head of the data specification into the component s-init of <u>CI</u> (cf. Fig. 12.22). From there the "initial format list" will be taken for format list expansion (cf. 12.6.2.2). The second action is the expansion of the data list of the head of the data specification which additionally depends on the transmission parameter. After completion of the expansion, the same action will be repeated with the tail of the data specification until all the data specification is worked up.

List- and data-directed data specifications are checked as to whether they are related with a file union the base of which is CST or CHAR. The <u>stream base</u> of a file union is BIT or CHAR (if the file union has been created by a bitstring or string option), or it is PRT, CST or BST in the other cases of file unions which contain the attributes PRT, CST (but not PRT) or BST, respectively.

If the check on the stream base is satisfied then the expansion of the data list is started in case of list-directed data specifications, and in the case of data-directed output data specifications. A data-directed input data specification will not be expanded (cf. 12.6.4).

<u>12.6.2.1 Data list expansion</u>

The data list which is a list of controlled or simple data items (cf. Fig. 12.18) is expanded into its scalar components. This process uses the expansion of aggregate expressions, and the expansion of controlled do-groups (the structuring of a controlled data item is similar to the structuring of a controlled do-group).

----

1) The type CHECK-DATA is used in connection with check standard system action, the empty type is used where no other type would be reasonable (cf. 12.6.3.2)

In case of stream output transmission, any resulting scalar expression is evaluated and transmitted (cf. 12.6.3).[1] If a data transmission has really occurred, and if transmission was over a file then

(1) the current value of the component "count" in the file union is incremented by one (cf. Fig. 4.11), and

(2) any transmission error flag in the FD-entry is inspected, and the transmit on-condition is called if necessary (cf. 12.6.3.1).

In case of stream input transmission the generation of any resulting scalar reference is evaluated and the next data field is transmitted (cf. 12.6.4).[1] If the data field should not be skipped then the assignment to the generation is performed, and if transmission was over a file, then the above steps (1) and (2) will occur in addition.

12.6.2.2 Format list expansion

The expanded format list is local to an edit-directed put or get statement. Hence, it is appropriate to store the current expanded format list in a component of the control information CI (s-expand in Fig. 12.22).[2]

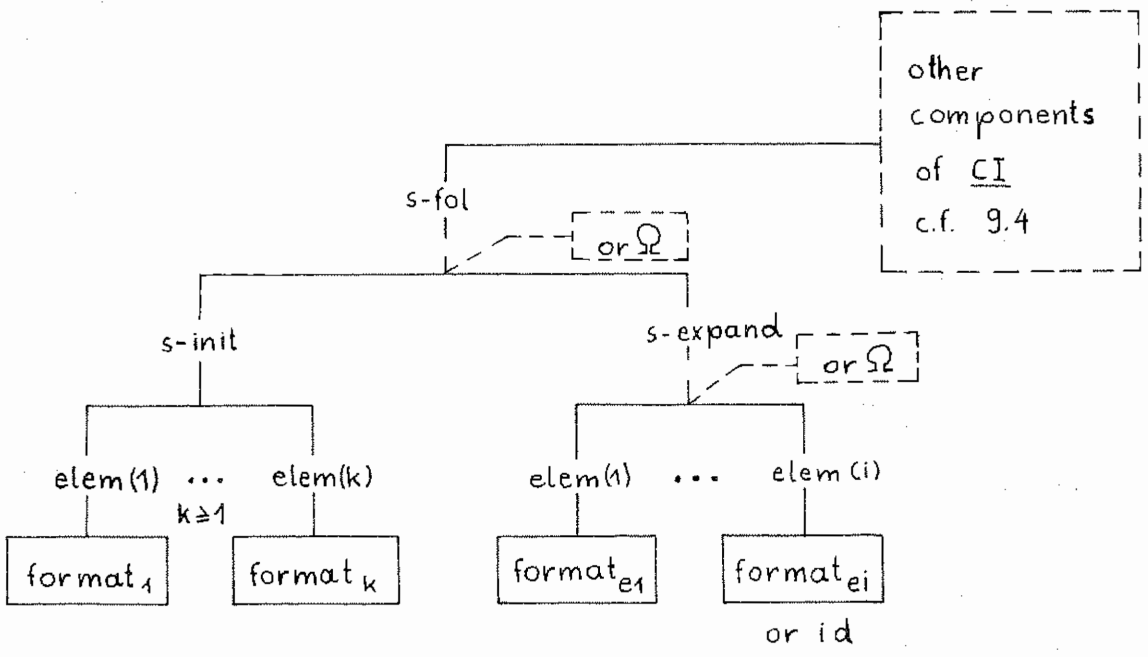


Fig. 12.22 Initial and expanded format lists of CI

---

1) Transmission is preceded, in the edit-directed case, by a request for the next evaluated simple data format, i.e., by the expansion of the format list and the interpretation of all intervening non-data formats.
2) In fact only the stacking and unstacking of CI at block boundaries is needed in connection with format list expansion.

It has been described at the beginning of 12.6.2 that the complete format list of the element of the data specification under consideration is contained in the component s-init, and that the component s-expand is initially empty.

The format list expansion is always activated by a request for the next evaluated simple data format (cf. Fig. 12.19). Such a request will cause a change of s-expand but it will never change s-init. The actions are easily explained if they are split up into an expansion step and a data format step which are taken iteratively.

The <u>expansion step</u> yields the next evaluated simple data or control format or empty depending on the following mutually exclusive case distinctions:

(1) If the expanded format list is empty or the empty list then it is replaced by the initial format list (wrap-around).

(2) If the expanded head [1] is an <u>iterated format</u> (cf. Fig. 12.19) then the resulting expanded format list consists of as many copies of format lists of the iterated format as the repetition factor indicates, concatenated with the expanded tail.[2]

(3) If the expanded head is a simple but <u>remote format</u> then the resulting expanded format list consists of the remote format list, concatenated with the remote format identifier (see below), and concatenated with the expanded tail.

(4) If the expanded head is a remote format identifier then the resulting expanded format list is the expanded tail.

(5) If the expanded head is a simple but <u>control</u> or <u>data format</u> then the format is checked for internal consistency, and for consistency with the stream base of the file union. If the check is positive then the format is evaluated and returned. The resulting expanded format list is the expanded tail.

In case (1) to (4) empty will be returned.

The <u>data format step</u> is performed one or more times as long as the expansion step yields empty or a control format. In the first case the expansion step is repeated immediately, in the second case it is repeated after executing the data transmission corresponding to the control format.

Steps (3) and (4) deal with remote formats. A remote format contains a reference which is evaluated, and the value must denote a format (label) constant. In addition, the block activation name and the condition prefix part of the format constant denotation must be the same as the current block activation name <u>BA</u> and the statement prefixes s-sp(<u>CS</u>), respectively. The <u>remote format list</u> contained in the format constant denotation is considered valid only if the <u>remote format identifier</u> of the format constant denotation is not yet contained in the expanded format list. This check caters for recursive usage of one and the same remote format list.

---

1) "Expanded head" is used for "head of the expanded format list, i.e., head•s-expand•s-fol(<u>CI</u>)".

2) "Expanded tail" is used for "tail of the expanded format list, i.e., tail•s-expand•s-fol (<u>CI</u>)".

12.6.3 STREAM OUTPUT

List-directed and data-directed transmission essentially consits of a conversion to character string type, and of the construction of names in character string form. The resulting string is handled by elementary data field transmission.[1]

Edit-directed transmission governed by data formats is treated similar to the above types of transmission. If edit-directed transmission occurs over a file and by control formats then elementary data transmission may occur immediately. All control formats directed to a print file may cause increase of the current line component of the file union ("line" in Fig. 4.11) until the pagesize "psz" is just exceeded by one. Only a page control format may reset line to one.

12.6.3.1 Elementary transmission

Elementary data field transmission depends on the transmission parameter and on a bit or character string. If the target is a string characterized by a file union corresponding to a string-put then the assignment of the string occurs element by element as long as the target may accomodate the source elements. This occurs in parallel with the incrementation of the current position s-hi (cf. Fig. 12.20). The error on-condition is called if proper assignment is impossible.

If the target is a data set then elementary data transmission will occur element by element with the current column properly updated. This updating is an incrementation of the component "col" of the file union until the linesize "lsz" (cf. Fig. 4.11) is just exceeded. Transmission of a data element will be preceded by the execution of a skip control format if col cannot be incremented anymore. This resets col to one.

Elementary data transmission is performed essentially in two steps which might be executed iteratively:

(1) Basic data transmission. This is the action stream-transmission (cf. Fig. 4.17) which uses the basic data transmitting function write (cf. 4.3.4.3). The third argument of the function (el) is a proper stream output or stream output print data element as enumerated in 4.3.3.1. The data set is replaced in ES by the data set yielded by application of the function write to its arguments, and the information returned is the unusual situation END or empty. In addition, a transmission error flag of the data set is deleted from the data set and copied into the FD-entry (cf. 4.3.4.4, 12.6.2.1).

(2) In case the unusual situation END has occurred previously, data set label processing and data set switching is performed with condition type EOV-BOV (cf. 12.4), and step (1) is retried if no transmission error occurred through data set label processing.

12.6.3.2 Special cases of stream tansmission

The copy action, and check standard system actions cause output to a standard system print file (cf. Fig. 12.1).

The copy action is elementary data field transmission of single

---

1) Eventually preceding tabulation is described in 12.6.3.2

characters or bits read [1] with a transmission parameter having an empty type. The action is preceded by a direct (hence implicit) opening of the standard system print file.

The check standard system action is expansion of the reference under consideration (cf. 12.6.2.1), and data-directed output of its components. There are some modifications of data-directed transmission which are indicated by a transmission parameter having the type CHECK-DATA. Before data transmission takes place the standard system print file is opened and the "count" component of the file union is initiated to zero.

The endpage standard system action executes a page format on that print file which is accessible by the file name being the argument of the action.

Elementary data field transmission over a print file is preceded by tabulation in case of list- or data-directed output. If tabulation is in the current line then elementary data transmission of the data element TABL will occur with the current column property adjusted, otherwise tabulation is in the next line. This means execution of a skip format followed by elementary data transmission of TABL with the current column adjusted to the very first tabulator position.

12.6.4 STREAM INPUT

Input of a single data field is performed in two steps: A scanning step and a conversion step. Depending on the data read in the first step, the second step may be skipped.

12.6.4.1 Scanning step

The scanning step is in fact composed of subsequent scannings, each with different arguments. In case of edit-directed input the scanning step is degenerated into simple counting which does not depend on the data read. Scanning plays the role of elementary data field transmission of stream output with the natural difference that it depends on the transmission parameter and the scanning argument (see below). It is also an inherent difference that scanning returns the data field which is a bit or character string (if edit-directed), or a character string optionally ending with the special elementary object ENDM-SCAN (if list- or data- directed). Hence, scanning is to some degree merely the reverse process of elementary (output) transmission described in 12.6.3.1.[2]

The scanning argument (Fig. 12.23) is of particular interest in list- and data-directed transmission.

----

1) Bits are converted to characters.
2) Elementary (input) data transmission may involve the copy action (cf. 12.6.3.2).

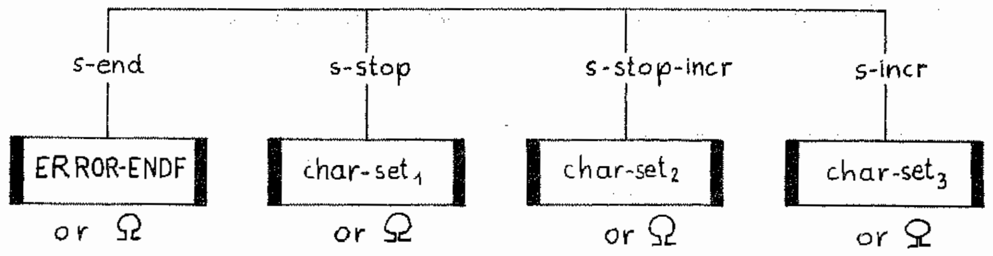


Fig. 12.23 Scanning argument

The components of the scanning argument have the following meaning:

(1) s-end. An on-condition call will be executed if the end of the input stream has been reached, and if the component is ERROR-ENDF. The endfile on-condition is called if transmission is over a file, the error on-condition is called otherwise.

If the component s-end is empty, scanning will terminate without raising any on-conditions. In this case ENDM-SCAN will be appended to the data field yielded in order to have an indication for this kind of termination.

(2) s-stop, s-stop-incr, s-incr. These components define the scanning classes the meaning of which is given below.

All scanning arguments actually used have the additional properties that no character occurs in more than one component, and that at most one of the components s-stop, s-stop-incr, and s-incr is empty.[1] This allows the formation of three <u>scanning classes</u> C(s-stop), C(s- stop-incr), and C(s-incr) for any scanning argument by the following rule: If the corresponding component of the scanning argument is empty then the class is the set of all characters not contained in the other components, otherwise it is the component itself.

The scanning classes have the following meaning:

(1) C(s-stop). If a character is scanned which belongs to that class then the scan is stopped. This is a case of normal termination.

(2) C(s-stop-incr). Same as (1), and in addition the current position will be incremented by one.

(3) C(s-incr). If a character is scanned which belongs to that class then the current position will be incremented by one, and the scan is continued.

The particular scanning argument of Fig. 12.24 defines the class C(s-stop) which is the empty set, C(s-stop-incr) which is {APOSTR, COMMA, BLANK}, and C(s-incr) which is the set of all characters except those contained in C(s-stop-incr). In other words: Scan as long as no

---

1) Besides that, s-stop and s-stop-incr are mutually non-empty.

apostrophe, comma, or blank is read. Any subsequent scan will start at the position following the apostrophe, comma, or blank.

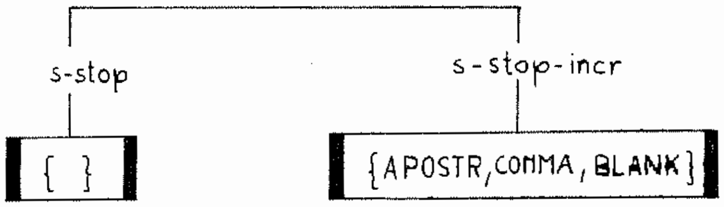


Fig. 12.24 Example of a scanning argument:
Second scan for list directed input

12.6.4.2 Conversion step

The conversion step checks the data field for syntactical correctness and builds an operand with the apparent attribute and value of the data field. In apparent attribute anc value of the data field. In case of data-directed input also the target reference is constructed which is checked against the data specification (the subsequent actions are as described in 12.6.2.1 under the paragraph stream input transmission).

12.7 MESSAGE TRANSMISSION

Corresponding sections of /5/:

3.6.4 The message part <u>M</u>

11.8 Display transmission

The following abbreviation is used in this section:

gen	generation or pseudo-generation

Message transmission is data transmission to and from the message storage (or message part) <u>M</u> effectuated by display statements by certain standard system on-units (cf. Fig. 12.1). It should be noted that the on-check standard system action performs output by a standard system print file, i.e., output goes to <u>PS</u> and not to <u>M</u> (cf. 12.6.3.2).

12.7.1 MESSAGE STORAGE

The message storage M (Fig. 12.25) is a global state component which serves to accumulate three types of messages in the components s-display, s-reply, and s-comment. All components are initially empty lists which are incremented on their tails by comments or named messages during interpretation of the program.

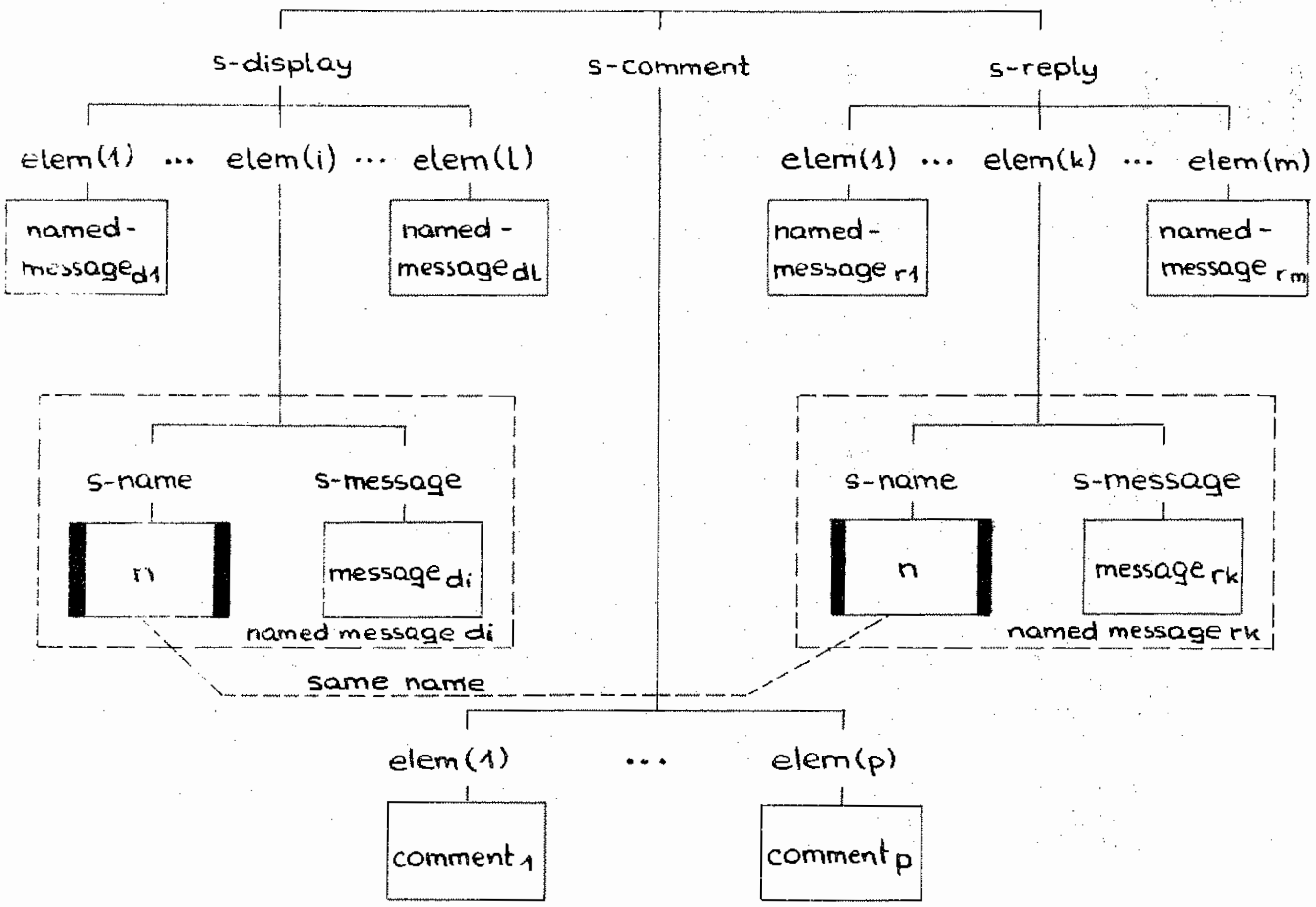


Fig. 12.25 Message storage M

The structuring of comments put out by certain standard system on-units is implementation-defined.

Named messages transmitted by display statements contain the message which is a list of character values and a unique name.

12.7.2 DISPLAY TRANSMISSION

The display, reply, and event options of a display statement are evaluated as described in 12.1. The resulting evaluated text et has the form (a) or (b) of Fig. 12.26.

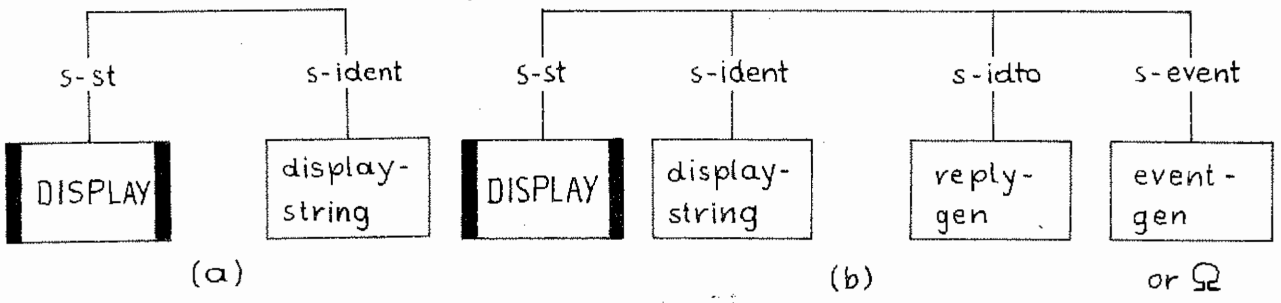


Fig. 12.26 Evaluated text of display statement

The form (a) having only a display option (s-ident), transmits the message (display-string) together with a newly created unique name to $\underline{M}$ (i.e., named-message$_{di}$).

The form (b) of Fig. 12.26 may cause the attaching of an I/O-event if the event option (s-event) is not empty. The relevant actions can be derived easily from 12.5.1. The presence of the reply option (s-idto) indicates that after the transmission of display-string (see above) a wait should occur as long as there is no matching named message in the component s-reply($\underline{M}$).

Environmental influences may append named messages to the list s-reply($\underline{M}$), i.e., after some time there might be the situation of matching messages named-message$_{rk}$ and named-message$_{di}$ shown in Fig. 12.25. In this case a wait entered after transmitting named-message$_{di}$ might be completed after assigning the message part of the reply message named-message$_{rk}$ to the target generation reply-gen.

Since this assignment may be executed as part of an I/O-event, on-condition calls must be avoided. This is similar to the assignment to a keyto option (cf. 12.5.3.5).

13. BUILT-IN FUNCTIONS AND PSEUDO VARIABLES

Corresponding sections of /5/:

12.2 Evaluation of built-in function references

12.3 Aggregate attributes of built-in functions

12.5 Table of built-in functions

12.6 Evaluation of the individual built-in functions

12.7.1 Assignment to pseudo variables

13.1 BUILT-IN FUNCTIONS

Built-in functions may occur as references in any expression context. They consist of an identifier, characteristic of the individual function to be applied, and an argument list. Evaluation of a reference to a built-in function returns an operand.

In /5/, the evaluation of a reference to a built-in function is accomplished in two steps: A general step, the effect of which can be described in a way common to all built-in functions and an individual step which is specific to the individual built-in function and returns the result operand.

(1) general step

The result of this step is a new argument list. Depending on the individual built-in function and the argument position, each element of the original argument list is converted to one of the following types and then inserted into the corresponding position of the new list.

operand: An operand is evaluated from the original argument and converted to a target attribute characteristic of every built-in function and argument place.

operand list: It can only occur if the original argument is an aggregate expression; operands are computed from its scalar elements and arranged as a list.

integer: From the original argument expression an operand is evaluated and converted into an integer constant.

evaluated aggregate attribute: In general, the original argument is an aggregate expression whose attributes are to be determined.

generation: The generation of the original argument is evaluated.

text: The original argument is inserted unchanged into the new argument list.

(2) individual step

According to the nature of the individual built-in function, the result operand is evaluated from the new argument list which was generated by the first step.

A similar procedure, devided into a general and an individual step, is followed in the evaluation of aggregate attributes of built-in function references.

Example:

Consider the following section of a program:

```
    ...
DCL B BIN FIXED(5),
    N FIXED,
    A BIT(6);
    ...
    N = 3;
    B = 23;
    A = BIT(B,N);
    ...
```

The built-in function BIT converts the first argument to a bit string and pads or truncates it according to the length specified by the second argument.

The reference BIT(B,N) is interpreted in the following way(cf. 12.2, 12.5, 12.6.2.1 of /5/):

(1) General step:

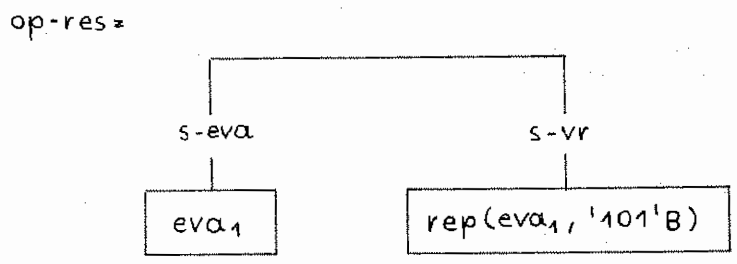
The actions of the first step are controlled by a table (cf. 12.5.2 of /5/. The following part of the table is used to illustrate, how it governs the argument evaluation:

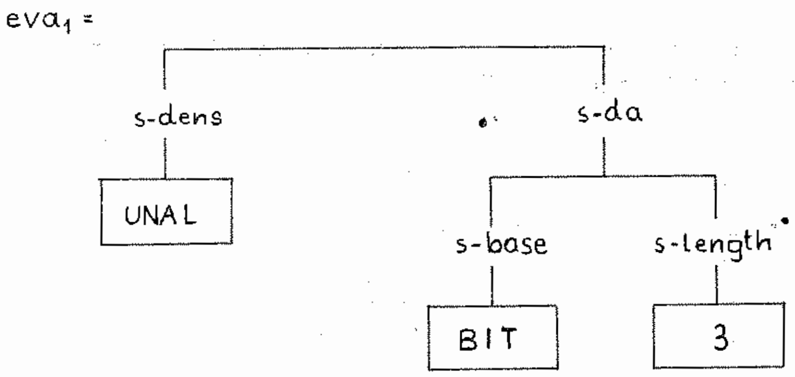
```
BIT  1 | ... |  OP   | STRING-EDA | ... | eval-bit(op₁,k₂)

     2 | ... | INTG |      *      | ...
```

For the first argument, OP causes an operand to be computed from B, and the result of this operation is then converted to string-type according to the entry STRING-EDA; let this final operand be op_1. INTG effects that an integer , k_2, is computed from N.

(2) Individual step:

The second step then begins with the execution of the instruction <u>eval-bit</u>(op_1,k_2). It returns the following operand op-res:





op-res is assigned to A; after assignment A has the value '101000'B.

13.2 ASSIGNMENT TO PSEUDO VARIABLES

Pseudo variables are means of accessing various storage parts which are otherwise inaccessible, for example: storage parts reserved for special purposes are accessible by condition- and multitasking pseudo variables; subparts of a scalar string may be assigned via the pseudo variable SUBSTR.

A reference to a pseudo variable occuring in the left part of an assignment statement consists of the name of the pseudo variable and a list of arguments. On evaluating the reference a pseudo generation is formed. A pseudo generation contains all information necessary to make the assignment. It consists of the name of the pseudo variable and a list of evaluated arguments which are either generations or integer values (depending on the type of the pseudo variable).

The pseudo assignment is carried out using

(1) the evaluated pseudo generation

(2) the operand which resulted from the evaluation of the right part expression.

The assignment includes conversion of the operand with target aggregates depending on the pseudo generation.

Example:

Let A be declared as

```
DCL A CHAR(5) INIT('12345');
```

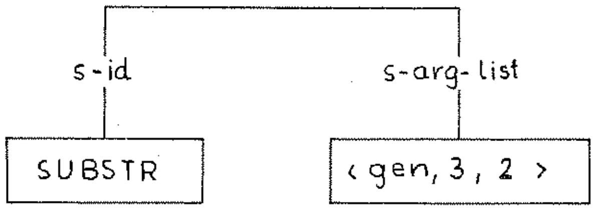
Consider the pseudo assignment

```
SUBSTR(A,3,2) = 'AB';
```

After execution, the value of A will be the string

'12AB5'.

The pseudo generation corresponding to the left part of the above pseudo assignment is



where gen is the generation of A.

On assignment, an operand is computed whose aggregate attributes are the same as those of gen, and whose vr-part is the value representation of '12AB5'. This operand is then assigned to gen.

14. OPTIMIZATION

Corresponding chapter of /5/:

13. Optimization

Optimization rules are introduced in the PL/I language to enable a compiler to produce more efficient object code. The definitions of these rules, however, are given in terms of values found at execution time, i.e., the formal description is based on the computation of a program.

There are two kinds of optimization rules in PL/I. First, the language definition has been relaxed so that expressions may be commoned. Second, attributes are added to the language which enable a compiler to do more efficient program optimization.

Both optimization rules are described formally by modifying the set of strict computations of a program (cf. 6.). To describe the rules for commoning of expressions, the concept of computation is extended, so that additional valid computations may be derived.However, the set of strict computations is reduced by rejecting computations which are invalid because of the wrong use of optimization attributes in the interpreted program text.

14.1 RULES FOR COMMONING OF EXPRESSIONS; THE REDUCIBLE ATTRIBUTE

To describe the rules for commoning of expressions (including the definition of the REDUCIBLE attribute), the concept of computation is extended in such a way, that it includes the possibility of steps in which expressions are commoned.

Such a step may occur in the course of a computation, if an instruction for evaluating an expression is ready for execution, and if another expression which is common with this expression has been evaluated previously during the computation. In this case, instead of evaluating the expression, the value derived from the earlier evaluation may be taken.

Although only scalar expressions may be commoned in a step of a computation, the notion of common expressions must be defined for the general case of aggregate expressions, since these may appear as arguments of common function references.

Two <u>expressions</u> are <u>common</u>, if they have the same structure, and if corresponding components are common references, the same constants or the same isubs.

For the definition of common references, if one of the references is a generic reference, it is replaced by the reference constructed from the selected entry reference by concatenating its argument part with the argument part of the original reference.

Two <u>references</u> are <u>common</u>, if they have the same evaluated list of name qualifiers, if their subscript lists are common (i.e. if corresponding subscript expressions are common), and if, according to the kind of the references, one of the following conditions holds:

(1) Both are refernces to variables, these variables are common, and, for entry components, the corresponding entry reference is reducible and the argument parts are common.

(2) Both are entry references to the same function, the reference in question is reducible, and the argument parts are common.

(3) Both are references to the same reducible builtin function, and the argument parts are common.

(4) Both are references to label, format or file constants with the same denotation.

Two _variables_ are _common_ in the following cases:

(1) Both are proper variables with the same generation and the same value representation.

(2) Both are defined variables with the same denotation, their bases are common references, and their positions are either empty or common expressions.

(3) Both are based variables with the same generation and the same value representation, their pointer qualifiers are common references, and their refer options have the same evaluated list of name qualifiers.

Note that based variables can not be commoned if they occur in argument expressions of function refernces, since in this case their generations are unknown in the states in question.

Whether an _entry reference_ is _reducible_ or not depends on the context of the entry reference (entry or non-entry), and, in the case of an entry context, it depends on the length of the argument part. An entry reference in an entry context is reducible, if either the argument part is empty, or the corresponding entry attribute is declared as reducible and the number of its reducible declared return types is at least equal to the length of the tail of the argument part. An entry reference in non-entry context is reducible, if the corresponding entry attribute is declared as reducible and all its return types of type entry are declared as reducible.

Note that the explained notion of reducible entry references constitutes the _definition_ of the _REDUCIBLE attribute_.

Two _argument parts_ are _common_, if corresponding arguments are common expressions and their descriptor aggregate attributes (given explicitly or by default) are essentially equal.

Example:

```
        ...

    P:PROCEDURE(X)  REDUCIBLE RETURNS
                    (ENTRY IRREDUCIBLE RETURNS (FLOAT));
                    ...  ; RETURN(Q); END;

    Q:PROCEDURE; ...  ; END;

      DECLARE F,G ENTRY VARIABLE, A,B,Y FLOAT;
      ...
      F = P(2*Y + 3);       }
                            }  common
      G = P(3 + 2*Y);       }
      ...
      A = P(2*Y + 3);       }
                            }  not common
      B = P(2*Y + 3);       }
      ...
```

According to the RETURNS attribute of the procedure P, a reference to P is reducible in entry context, but not in non-entry context. I.e., the first two references to P are common, the second two are not.

14.2 THE REORDER ATTRIBUTE

The REORDER attribute may be specified for a block or procedure body if it is required that the execution of the body should give the same results as execution of the body according to the strict definition of PL/I unless there is a computational or system action interrupt during the execution of the body. The definition of the REORDER attribute is given by its relation to the interrupt handling facilities of PL/I.

The formal definition consists in testing whether individual members of the set of strict computations of a program are erroneous because of the wrong use of the REORDER attribute, i.e., if the REORDER attribute associated with a body leads to undefined situations due to on-units executed during the computation of this body.

To be precise, a computation is erroneous because of the wrong use of the REORDER attribute in the interpreted program text, if the following conditions hold:

(1) There exists a section $<\xi(i1),\ldots,\xi(i2)>$ as part of the computation, called reorder-section, which constitutes (disregarding steps belonging to another task) the computation of a block or procedure body declared with the REORDER attribute.

(2) There exists a section $<\xi(j1),\ldots,\xi(j2)>$ as part of this reorder-section, called on-section, which constitutes the computation of an on-unit.

(3) There exists a pointer p which is contained in the allocation state of the storage between $\xi(k1)$ and $\xi(k2)$ (exclusively), and which does not belong to an automatic variable declared in a block

or procedure whose body is declared with the ORDER attribute and computed inside the reorder-section.

(4) There is a reference to this pointer p in a state $\xi(k)$ between $\xi(k1)$ and $\xi(k2)$ which is not guaranteed under such circumstances.

A <u>reference</u> to the pointer p in the state $\xi(k)$ is <u>not guaranteed</u> under the above circumstances, if one of the following alternatives holds:

(a) $\xi(k)$ lies in the on-section.
p is not referenced by the use of on-builtin functions.
p is allocated or freed or its content is modified in the reorder-section (but outside the on-section), or p is allocated outside the on-section (possibly also outside the reorder-section) and belongs to a controlled variable which is allocated or freed in the reorder-section.

(b) $\xi(k)$ lies in or after the reorder-section.
p is allocated or freed or its content is modified in the on-section without the use of on-pseudovariables, or p is allocated before the end of the reorder-section and belongs to a controlled variable which is allocated or freed or modified in the on-section without the use of on-pseudovariables.

(c) $\xi(k)$ lies in or after the reorder-section.
There is an abnormal return from the on-section.
p is allocated or freed or its content is modified in the reorder-section, or p is allocated before the end of the reorder-section and belongs to a controlled variable which is allocated or freed in the reorder-section.

Examples:

ad (a):

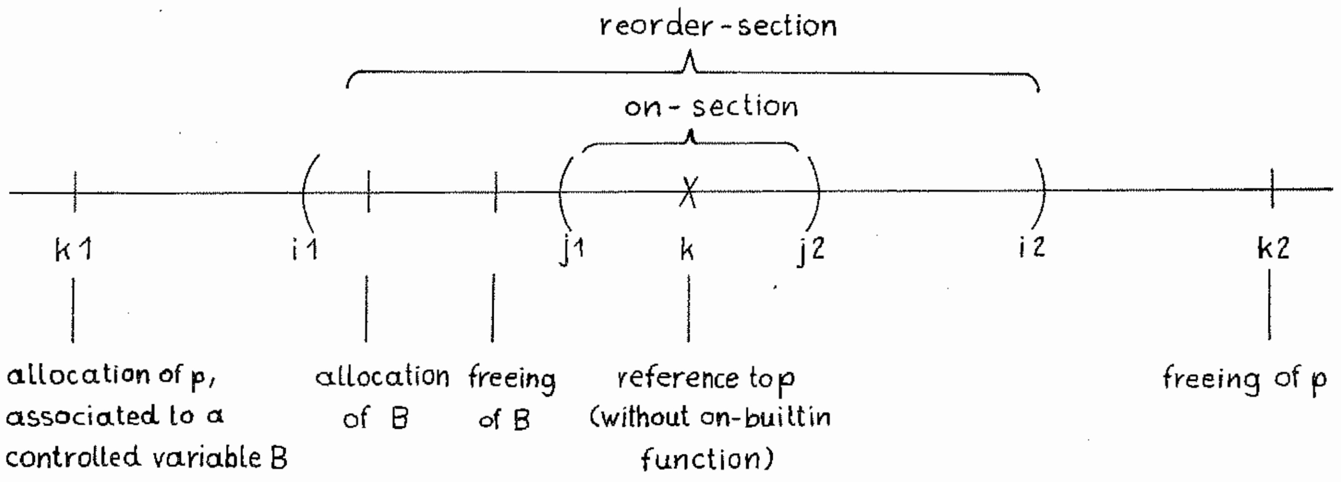

reorder-section
on-section
k1
i1
j1
k
j2
i2
k2
allocation of p, associated to a controlled variable B
allocation of B
freeing of B
reference to p (without on-builtin function)
freeing of p

Fig. 14.1

The reference to p is not guaranteed, since, due to the reordering process, the on-section could fall between allocation and freeing of B in the reorder-section.

ad (b):

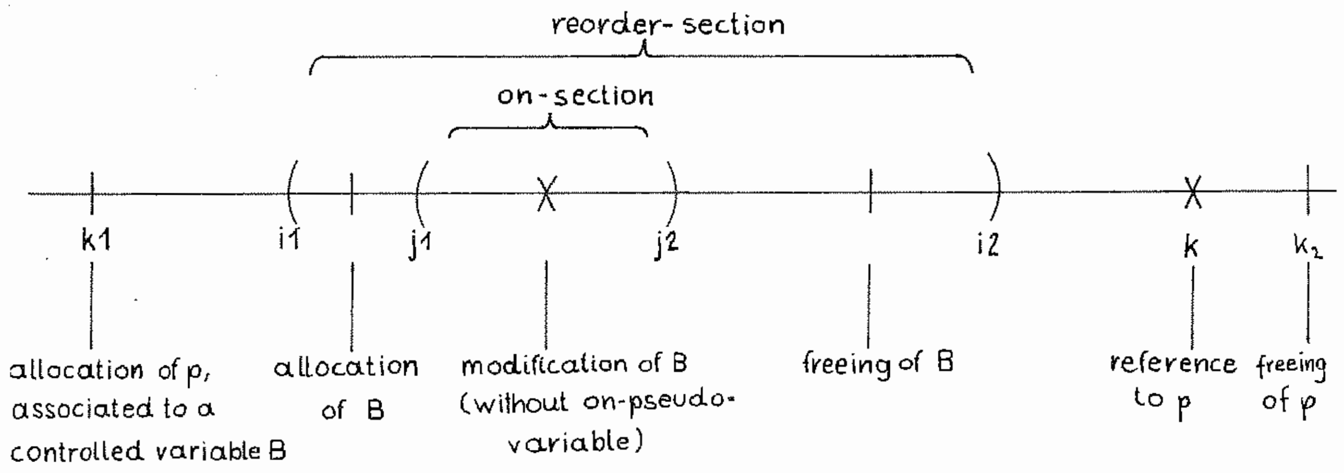

reorder-section
on-section
k1
i1
j1
j2
i2
k
k2
allocation of p, associated to a controlled variable B
allocation of B
modification of B (without on-pseudo-variable)
freeing of B
reference to p
freeing of p

Fig. 14.2

The reference to p is not guaranteed, since, due to the reordering process, the on-section could fall between freeing of B and i2.

ad (c):

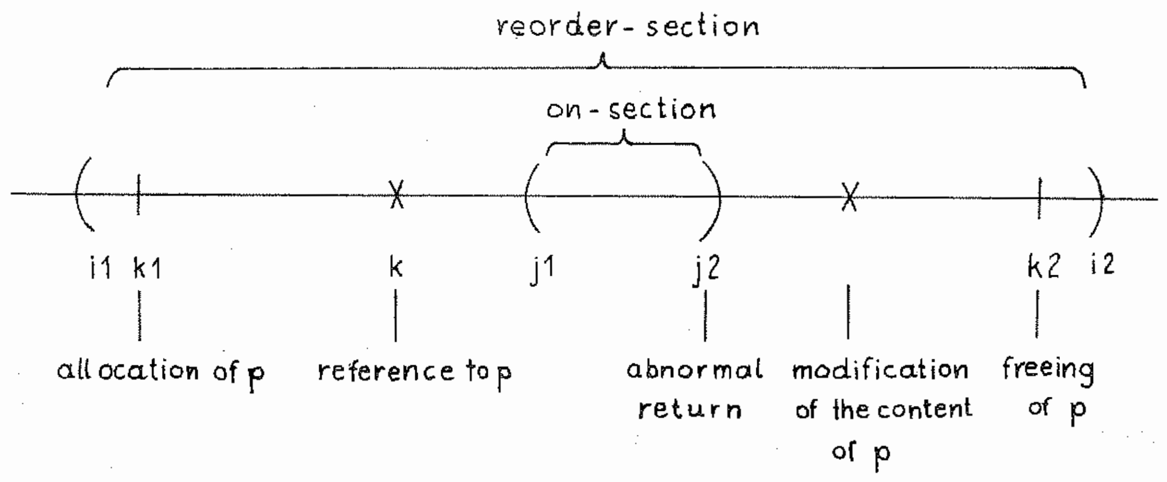


Fig. 14.3

The reference to p is not guaranteed, since, due to the reordering process together with the abnormal return from the on-section, the reference to p and the modification of the content of p could be interchanged.

14.3 THE RECURSIVE ATTRIBUTE

A computation is erroneous because of the wrong use of the RECURSIVE attribute in the interpreted program text, if it contains the computation of a procedure which invokes itself though it is not declared with the RECURSIVE attribute.

APPENDIX: GLOSSARY

This glossary is a compilation of technical terms used in the document. The section quoted in the right column refers to the place where the term is explained. In the document, this place is emphasized by underscoring.

The primary intent of the glossary is to serve as an aid for reading the document. Occasionally, terms are characterized by a qualification added in parentheses in order to delimit the scope of the reference.

abnormal termination (of a task)	7.4
abstract identifier	2.2
abstract program	1
abstract syntax	1
abstract syntax of the states	1
active (block activation)	8
active (inner data set)	4.3.3.2
active (task/event)	7
aggregate attribute	2.2.1
aggregate attribute of a reference	10.2
aggregate name	5.3
allocate	5.3
allocate statement	2.3
allocation (of a buffer)	12.5.3.2
allocation state	4.2.3
applicable	4.2.1
argument expression	8.3.1
argument part (of a reference)	2.4.1
array	2.2.1
assignment	5.3
associated event variable	7.2
associated task variable	7.2
asynchronous interrupt	11.4.1
attaching (a task)	7.3
attention	12.2.2
attention (declaration)	2.2
attention condition call	11.4.1
attention directory	11.1.1
attention enabling state	11.1.2
attention environment directory	11.1.2
attention event	12.2.2
attention identification	11.1.1
attention information	11.1.1
based free set	7.2
based variable	2.2
basic data transmitting action	4.3.4
basic data transmitting function	4.3.4
begin block	2.1
bit string value	4.1.1
bit to character conversion	10.3.2
bit to numeric conversion	10.3.2
bit value	4.1.1

block	2.1
block activation	8
block activation name	8.2.2
block activation type	8.2.4
block local	3.2
buffer	12.5.3.1
builtin function	2.2
character string value	4.1.1
character to bit conversion	10.3.2
character to numeric conversion	10.3.2
character value	4.1.1
check standard system action	12.6.3.2
close element	12.1
comment	12.7.1
common argument parts	14.1
common expressions	14.1
common references	14.1
common variables	14.1
complete set of attributes	12.2.3.1
completion value	7.5
computation	1
computation step	6.2
concrete identifier	2.2
concrete program	1
concrete syntax	1
condition action	11.3.2
condition builtin function part	11.5
condition call	11.4.1
condition indication	12.5
condition part	2.3
condition prefix part	2.1
condition selector	11.1.5
condition state	11.1.4
condition type	12.4
conditional unlocking	12.5.2
connected (flag of a parameter declaration)	2.2.1
connected (generation)	4.2.5
constant	2.4
control format	12.6.2.2
control information <u>CI</u>	9
controlled group	9.5
controlling variable	9.5
conversion between pointers and offsets	4.2.7.2
copy action	12.6.3.2
cross-section	4.2.6
current (block activation)	8
current generation	5.3
current task	3
current task-event name	7.2
data attribute	2.2.1
data format step	12.6.2.2
data set	4.3.1
data set directory	4.3.1
data set label	4.3.3.1
data set name	4.3.1
data set switching	12.4
data set title	4.3.1
data transmission	4.3.1
daughter task	7.3

declaration	2.2
declaration part	2.1
defined variable	2.2
definition of the REDUCIBLE attribute	14.1
density	2.2.1
direct (access to standard system print file name)	12.2.1
dummy operand	8.3.1
dynamic descendant (block activations)	8
elementary allocation	4.2.3
elementary data field transmission	12.6.3.1
elementary data transmission	12.6.3.1
elementary freeing	4.2.3
end state	1
endpage standard system action	12.6.3.2
entry attribute	8.3
entry constant	2.2
entry denotation	8.3
entry identifier	2.1.3
entry part/point	2.1.3
environment attribute	4.3.1
environment step	6.2
epilogue (of a block activation)	8.2.4
erroneous computation (by wrong use of RECURSIVE attribute)	14.3
erroneous computation (by wrong use of REORDER attribute)	14.2
evaluated aggregate attribute	2.2.1
evaluated attention condition	11.1.5
evaluated condition	11.1.5
evaluated statement text	12.1
event variable	7
exactly representable	4.1.2
expansion step (of a format list)	12.6.2.2
explicit closing	12.3.1
explicit opening	12.2.2
explicit picture attribute	10.3.3
fd-status (file directory status)	12.2.1
file	5.5
file constant	2.2
file name	5.5
file union	4.3.2
file union name	5.5
file value	5.5
file-get (statement)	12.6.1
file-put (statement)	12.6.1
format label constant	2.2
free	5.3
free set	8.2.4
freeing (of a buffer)	12.5.3.1
fully qualified name	2.4.1
function denotation	8.2.4
generation	4.2.5
generation list	5.3
generic (family) member	2.2
generic identifier	2.2
generic selection	8.3.4
global	3.2
group	2.3
group (repeated execution)	9.5

position (of an inner data set)	4.3.3.1
pre-evaluation of expressions	10.2.1
prefix expression	2.4
prepass	1
priority	7.2
priority scheduler	7.1
procedure body	2.1
procedure body part	2.1
procedure call	8.3
program	2
program local	3.2
programmer named condition	2.2
proper closing	12.3.2
proper data element	4.3.3.1
proper data transmission	12.5.3
proper inner data set	4.3.3.1
proper interpretation	1
proper key	4.3.3.1
proper opening	12.2.3
proper statement	2.3
proper variable	2.2
pseudo generation	13.2
raising of a condition	11.5
record (data) element	4.3.3.1
reducible entry reference	14.1
reference	2.4
reference list	4.2.6
remote format	12.6.2.2
representable	4.1.2
representation	4.1.2
return type (of an entry point)	2.1.3
return type (of an entry declaration)	2.2.2
return type (used in epilogue)	8.2.4
scalar	2.2.1
scanning argument	12.6.4.1
scanning class	12.6.4.1
scope attribute	2.2.1
semi-complete (I/O-event)	7.5
share	5.11
sharing (of a data set)	4.3.3
size	4.2.1
standard system action	11.3.1
standard system print file name	12.2.1
statement	2.3
statement label constant	2.2
statement list	2.3
statement location	2.1.3
status (of a file union)	12.2.3.3
status value	7.5
storage class attribute	2.2.1
storage mapping function	4.2.4
storage overflow	4.2.3
stream (data) element	4.3.3.1
stream base	12.6.2
string-get (statement)	12.6.1
string-put (statement)	12.6.1
structure	2.2.1
sub-generation	4.2.6
subscript list (of a reference)	2.4.1