TECHNICAL REPORT

TR 25.083
28 June 1968

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

P. LUCAS
K. ALBER
K. BANDAT
H. BEKIĆ
P. OLIVA
K. WALK
G. ZEISEL

IBM LABORATORY VIENNA
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.
INFORMAL INTRODUCTION TO THE ABSTRACT
SYNTAX AND INTERPRETATION OF PI/1

I

This document provides an informal introduction to the syntax and application of F. The focus is to give sufficient information of the parts and elements of the formal definition so that knowledge of interest can be formulated and evaluated on the concepts and formal details.

Abstract

Index Terms for IBM Subject Index

PI/1
Formal Definition
Syntax, Semantics
Semantics
Programming

TR 52687
28 June 1980
PREFACE

This document is part of a series of documents which represent the formal definition of syntax and semantics of PL/I issued by 28 June 1968:

/1/ LUCAS, P., LAUER, P., STIGLEITNER, H.: Method and Notation for the Formal Definition of Programming Languages.


IBM Laboratory Vienna, Techn. Report TR 25.084.


The method and notation for 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.


An outline of the method is given in /1/, which 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 inside and outside IBM.
The language defined in this present version is PL/I as specified in the official PL/I Language Specifications Form No. Y33-6003 with the exception of the following features which are not included:

- optimizing attributes (they are included in the concrete syntax but not in the abstract syntax; they are only tested for compatibility with other attributes and used for implication of default attributes),
- implicit conversion between offsets and pointers,
- the REFER option,
- the implicit rules for ordering initializing actions in the prologues of blocks and procedures.

The draft for this document was completed by 15 January 1968. It has been subject to validation by members of the PL/I Language Department of IBM UK Laboratories Hursley, England. The results of the checking effort conducted in Hursley have been taken into account in this present corrected form.

The formal definition given here includes more details than are given in the Specifications. These details have been confirmed as far as possible by the PL/I Language Department Hursley during the validation process. Some amendments and clarifications to the Specifications were generated during this process and will be published as Technical News Letters to the Specifications.

Contribution to the document:

Project coordinator: P. Lucas
Authors: their main contributions are given by chapters
    K. Alber 2., 7.
    K. Bandat 6.
    H. Bekić 4.1, 8.3
    P. Lucas 5.
    P. Oliva 4.3, 9.
    K. Walk 4.2, 8.
    G. Zeisel 10.

Proofreading: G. Zeisel
Coordination of production: F. Schwarzenberger, H. Hoja, W. Pachl
Typing: H. Deim, W. Schatzl
Special graphics and diagrams: G. Lehmayr
Validation: C.D. Allen, P. Seaman
INTRODUCTION

This document is an informal introduction to the formal definition of the abstract syntax and interpretation of PL/I /6/. 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, datasets, storage ...) 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. Chapters 6, 7, 8, 9 and 10 explain the basic behaviour of the PL/I machine in interpreting the major components of a program, where by 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 the method see /1/). The terminology and style of 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 /6/. 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 $s_1$, $s_2$, ..., $s_n$ are selectors and $v_1$, $v_2$, ..., $v_n$ are the immediate components.
INTRODUCTION

This document is an initial introduction to the formalization of the

Introduction of abstract syntax and interpretation of SQL. In this chapter, we will
present the necessary information on the syntax and semantics of the formal

The conceptual part of this introduction is divided into three

with a brief overview of previous work on interpretation, with particular

In this chapter, we provide some background on

The chapter on abstract syntax and abstract semantics of

We summarize the main components of the SQL language, to

The chapter on abstract syntax and abstract semantics of

We summarize the main components of the SQL language, to

The chapter on abstract syntax and abstract semantics of

We summarize the main components of the SQL language, to

The chapter on abstract syntax and abstract semantics of

We summarize the main components of the SQL language, to

The chapter on abstract syntax and abstract semantics of

We summarize the main components of the SQL language, to

The chapter on abstract syntax and abstract semantics of

We summarize the main components of the SQL language, to
# CONTENTS

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

2. STRUCTURE OF ABSTRACT PROGRAMS  2-1
   2.1 Program  2-1
   2.2 Procedure Body  2-3
   2.3 Declarations  2-5
      2.3.1 Proper Variable Declarations  2-7
      2.3.2 Entry Declarations  2-14
   2.4 Statements  2-16
      2.4.1 Begin Block  2-17
      2.4.2 Group and Statement List  2-18
      2.4.3 Allocate Statement  2-18
   2.5 Expressions  2-20
      2.5.1 References  2-21
      2.5.2 Constants  2-23

3. THE STATE OF THE PL/I MACHINE  3-1

4. STORAGE AND DATA  4-1
   4.1 Representation of Data in the PL/I Machine  4-1
      4.1.1 Values, Value Representations, Operands  4-1
      4.1.2 The Transition between a Value and Its Representation  4-3
         4.1.2.1 The Non-Elementary Cases  4-5
         4.1.2.2 The Elementary Cases  4-6
   4.2 Internal Storage and Generations of Variables  4-8
      4.2.1 Storage and Storage Parts  4-8
      4.2.2 Elementary Assignment and Reference  4-9
      4.2.3 Elementary Allocation and Freeing  4-10
      4.2.4 Storage Mapping  4-12
      4.2.5 Generations of Variables  4-13
      4.2.6 Subgenerations of Generations  4-14
4.2.7 Survey of Attributes Depending on the Storage Model
4.2.7.1 Areas
4.2.7.2 Pointers, Offsets
4.2.7.3 The PACKED and ALIGNED Attributes
4.2.7.4 Cells

4.3 External Storage and Files
4.3.1 Organization of External Storage
4.3.2 Access to Titled Datasets
4.3.3 Dataset Mapping
4.3.3.1 Data Transmission from External Storage
4.3.3.2 Data Transmission to External Storage
4.3.3.3 Some Small Examples
4.3.4 Files and Datasets
4.3.4.1 The Notion of a File
4.3.4.2 Organization of the File Union Directory
4.3.4.3 Organization of the File Directory
4.3.5 Basic Data Transmission to and from a File
4.3.6 Access to a Titled Dataset by Different Files
4.3.6.1 Parallel or Shared Access
4.3.6.2 Successive Access

5. IDENTIFIERS AND THEIR SIGNIFICANCE
5.1 Declarations and References
5.2 Denotation and Attributes
5.3 Proper Variables
5.4 Based and Defined Variables
5.5 Files
5.6 Procedures
5.7 Generic Names
5.8 Builtin Functions
5.9 Labels
5.9.1 Labels which Serve as Designations of Goto Statements
5.9.2 Format Labels
5.10 Some Remarks

6. PARALLEL TASKS AND I/O-EVENTS
7. FLOW OF CONTROL

7.1 Program Initialization
   7.1.1 The Initial State of the PL/I Machine
   7.1.2 The Prepass

7.2 Block Activation
   7.2.1 The Dump
   7.2.2 Interpretation of a Begin Block
   7.2.3 Environment Updating
   7.2.4 The Epilogue Information

7.3 Sequential Interpretation of a Statement List
   7.3.1 General Mechanism
   7.3.2 Structure of the Control Information CI

7.4 Goto Statement
   7.4.1 The Denotation of a Label Constant
   7.4.2 Scanning a Statement List for a Label
   7.4.3 Goto within the Current Statement List
   7.4.4 Goto out of the Current Statement List

7.5 Procedure Call
   7.5.1 Argument Passing
   7.5.2 Function Reference
   7.5.3 Return from a Procedure

8. ALLOCATION, ASSIGNMENT AND EXPRESSION EVALUATION

8.1 Allocate Statement and Free Statement
   8.1.1 Allocate Statement
      8.1.1.1 Allocation of Controlled Variables
      8.1.1.2 Allocation of Based Variables in Main Storage
      8.1.1.3 Allocation of Based Variables in Areas
   8.1.2 The Free Statement
      8.1.2.1 Freeing of Controlled Variables
      8.1.2.2 Freeing of Based Variables in Main Storage
      8.1.2.3 Freeing of Based Variables in an Area
8.2 Assignment Statement, Expression Evaluation, Reference to Variables

8.2.1 Expansion of Aggregate Assignment Statement
8.2.2 Assignment Statement BY NAME
8.2.3 Scalar Assignment
8.2.4 Area Assignment
8.2.5 Assignment to Pseudo Variables
8.2.6 Expression Evaluation
8.2.7 Reference to Variables

8.2.7.1 Proper Variables
8.2.7.2 Defined Variables
  8.2.7.2.1 Isub-Defining
  8.2.7.2.2 Correspondence Defining
  8.2.7.2.3 Overlay Defining
8.2.7.3 Based Variables

8.3 Infix and Prefix Operations, Conversion, Numeric Pictures

8.3.1 Infix and Prefix Operations
8.3.2 Conversion
8.3.3 Representation and Evaluation of Numeric Pictures

9. INPUT AND OUTPUT

9.1 Opening
9.1.1 Central Part of Opening
9.1.2 Implicit Opening
  9.1.2.1 Opening with a File Identifier
  9.1.2.2 Opening without a File Identifier
9.1.3 Explicit Opening

9.2 Record I/O
9.2.1 Classification of Data Transmitting Statements
9.2.2 Actual Data Transmission
  9.2.2.1 Direct Files
  9.2.2.2 Unbuffered Files
  9.2.2.3 Buffered Files
9.2.3 I/O-Events
9.2.4 Locking and Unlocking
9.2.5 Remarks on Record I/O Statements
9.3 Stream I/O
  9.3.1 Get Statement
    9.3.1.1 Source Specification
    9.3.1.2 Data Specification
    9.3.1.3 Format List
    9.3.1.4 Data Fields
    9.3.1.5 Copy Action
  9.3.2 Put Statement
    9.3.2.1 Target Specification
    9.3.2.2 Check Standard System Action

9.4 Closing

9.5 Message I/O
  9.5.1 Message Storage M
  9.5.2 Display Statement
  9.5.3 Output of Comments

10. CONDITIONS

  10.1 The Condition State CS
  10.2 Enabling and Disabling for Conditions
  10.3 On and Revert Statement
  10.4 Condition Activations
  10.5 Condition Builtin Function Status
1. **THE OVERALL STRUCTURE OF THE FORMAL DEFINITION OF PL/I**

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

![Block Diagram](image)

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 some 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 1) and to remove some notational conventions 2). The result of the translator is an abstract object as described in /1/ 3) 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 trees considered.

1) e. g. the translator collects all declarations spread over one block into one component of the block.
2) e. g. partially qualified names are fully qualified by the translator.
3) i. e. a tree with named branches and elementary objects at the terminal nodes.
The rest of the interpretation is defined by the PL/I machine whose initial state \( S_0 \) is produced 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 /1/, 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 \( S_p \) contains then the abstract program modified according to (4).
Finally the proper interpretation interprets the prepassed abstract 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.
The proper interpretation interprets the program's syntax.

According to the meaning of the programming statement, the statement syntax of P/M may be taken as the center of the program definition in this sense. This process can be taken as the center of the correct line in P/M. I agree with a special representative position of P/M as a character string and the process to the right gears with the meaning of P/M.

Only the statement syntax and the interpreter are considered in this document.
2. STRUCTURE OF ABSTRACT PROGRAMS

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. In the following the term program is used instead of "abstract program". For the formal definition see chapter 12 of /6/. The following is the key to the abbreviations used in this chapter. Prefixes and suffixes have been omitted (e.g. the prefix s- indicating a selector).

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>al</td>
<td>incomplete data attribute specified in the allocate statement</td>
</tr>
<tr>
<td>arg</td>
<td>argument</td>
</tr>
<tr>
<td>cond</td>
<td>condition</td>
</tr>
<tr>
<td>da</td>
<td>data attribute</td>
</tr>
<tr>
<td>decl</td>
<td>declaration</td>
</tr>
<tr>
<td>den</td>
<td>denotation</td>
</tr>
<tr>
<td>dens</td>
<td>density</td>
</tr>
<tr>
<td>descr</td>
<td>description</td>
</tr>
<tr>
<td>eda</td>
<td>evaluated data attributes</td>
</tr>
<tr>
<td>elem</td>
<td>element</td>
</tr>
<tr>
<td>id</td>
<td>identifier</td>
</tr>
<tr>
<td>init</td>
<td>initial</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
<tr>
<td>lbd</td>
<td>lower bound</td>
</tr>
<tr>
<td>op</td>
<td>operand</td>
</tr>
</tbody>
</table>

2.1 Program

A program is an object consisting of a declaration part, declaring external procedure entries only, and the list of all its external procedure bodies (Fig. 2.1).
The structure of a declaration part and the relation between declarations of entry identifiers and the corresponding procedure bodies is described in 2.3, the structure of a procedure body in 2.2.

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.

Example:

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:

```
\begin{figure}[h]
\centering
\begin{tikzpicture}
  \node (decl_part) {s-decl-part};
  \node [below of=decl_part] (elem1) {\text{declarations of A,B,C,D,E}};
  \node [below of=elem1] (body_A) {body_A};
  \node [right of=elem1] (elem2) {\text{body}_D};
  \node [right of=elem2] (elem3) {\text{body}_E};
  \node [below of=elem2] (s-body-list) {s-body-list};

  \path[->] (decl_part) edge node {\text{elem}(1)} (elem1)
                    edge node {\text{elem}(2)} (elem2)
                    edge node {\text{elem}(3)} (elem3);
\end{tikzpicture}
\caption{Fig. 2.2}
\end{figure}
```

Note: A program may be thought of as an incomplete block (cf. 2.4.1) to which the external procedure bodies are local: Instead of a statement list the initial call statement is activated, which is specified apart from the program; and no conditions are "enabled" for this block (cf. 7.1.1).
2.2 Procedure Body

An (internal or external) procedure body is an object consisting of four components:

- **s-decl-part**: A declaration part collecting all declarations local to the procedure body (whether they are explicit, contextual or implicit in the concrete program); the structure of a declaration part is described in 2.3.
- **s-body-list**: A list of all those internal procedure bodies which are local to the procedure body and correspond to entry declarations contained in the declaration part (this correspondence is described in 2.3.2).
- **s-cond-part**: A condition part containing the condition "enabling" information given by the condition prefixes of the procedure body in the concrete program.
- **s-st-list**: A procedure statement list. All declarative information (declarations, procedures, format sentences), which is valid for the complete procedure body independently of its location in the concrete program text, is collected in the declaration part and body list. Therefore the procedure statement list lists only executable statements and entry points of the procedure body in their given order, including the main entry point. (Statements are described in 2.4.)

An entry point of a procedure body, occurring as an element of the procedure statement list, is an object consisting of: the entry identifier (for which the procedure statement list is scanned at procedure call), the parameter list (list of parameter identifiers) and the return type, i.e. the data attributes of the value to be returned if this entry point is used by a function reference.
If no data attributes are specified explicitly in the concrete program the default ones are taken into the abstract program. If an entry point in the concrete program has more than one entry identifier, for each of them an individual entry point occurs in the abstract program (their return types may differ).

Example:

The concrete procedure body:

```
(CONV):A:I:PROC(X,Y);

  statement 1;
  DCL X ...;
  C:EENTRY (X) FIXED;
  statement 2;
  DCL 1 Z ...;
  P:PROC ...; END P;
END A;
```

is translated into:

![Diagram of translation process](image)

Fig. 2.4
2.3 Declarations

Each block (i.e. begin block or procedure body) has as one of its components a declaration part. In this declaration part all declarative information is collected which is local to the block, except the bodies of the local procedures which are collected into a separate body list. This declarative information is all information valid for the whole block independently of 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 declaration, 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 rules from the concretely specified attributes are inserted by the translator.

To each identifier of the concrete program corresponds uniquely an abstract identifier which is an elementary object satisfying the predicate is-id. The transformation between the character string representing an identifier in the concrete program and its corresponding abstract identifier is performed by the function \( \text{mk-id}(\text{l-o-76(214) of }/6/) \). 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 \( \text{mk-id}(A) \)).

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

```
   id_1  id_2  id_n
   decl_1 decl_2 ... decl_n
```

Fig. 2.5 decl-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 procedure body given in the example in 2.2 has the following structure:

```
X
declaration
of X
```
```
Y
declaration
of Y
```
```
Z
declaration
of Z
```
```
P
declaration
of P
```

Fig. 2.6

Each individual declaration is an object, whose structure depends essentially on the type of the declaration. There are the following 10 types of declarations:

1) **Proper variables.** Their declarations are described in some detail in 2.3.1.

2) **Entry identifiers.** Their declarations and their correspondence to the procedure bodies are described in 2.3.2.

3) **File identifiers.** Their declarations consist of: the set of file attributes (those explicitly declared in the concrete program and those necessarily implied by them); the scope; the file identifier itself serving as title; the implementation dependent environment attribute.

4) **Defined variables.**

5) **Based variables.**

6) **Built-in function identifiers.** For each built-in function used, whether it is declared explicitly with the attribute BUILTIN in the concrete program or not, in the abstract program there is a declaration which is just the elementary object BUILTIN.

7) **Statement label constants.** They have the elementary object LABEL as declaration. Also, for subscripted statement labels in the concrete program, which serve as initial values for a declared label variable array, the translator creates declarations of label constants and inserts corresponding initial attributes into the declaration of the label variable array.
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 and the condition prefix part.

Generic entry identifiers. In an abstract program a generic declaration essentially is a set of generic references each of which consists of an entry identifier and a parameter descriptor list used for generic selection. The individual entry identifiers declared within a generic declaration in a concrete program have their own declarations independently in the abstract program, since they may also be invoked independently.

Programmer named conditions.

2.3.1 Proper variable declarations

The term proper variable denotes level-one variables, i.e. arrays, structures, cells or scalars which are not themselves components of arrays, structures or cells, if they are internal, external or parameter. Defined and based variables are not included.

The declaration of a proper variable consists of five components:

\[ \text{prop-var} \]

These components are:

a) 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.

b) The storage class attribute. Non-controlled parameter declarations have the null object \( \varnothing \) as the storage class component.

c) The data attributes as described below in more detail.
(d) The density attribute PACKED or ALIGNED.

(e) The initial set. In order not to burden the structure of data attributes (especially in the case of arrays), all initial attributes belonging to the components of a proper variable are collected into a separate initial set. To show which component is to be initialized by an individual attribute each element of this set consists of two components: the initial attribute itself and a reference to the initialized component. Also subscripted statement labels of the concrete program, serving as initial values for label variable arrays, lead to analogous elements of the initial set.

Example:

The concrete declaration

```
DCL X FIXED CTL INIT(0);
```

is translated into:

```
5-e.cope
5-'<>tg-cl s-dsa
dens:s-init-set
```

Fig. 2.8 Example of a declaration part

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

(a) 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 or cells is naturally handled in the same way with the only difference that its base elements are described as structures or cells. So, an abstract program has only one-dimensional array data attributes; the elements may be arrays, structures, cells or scalars.
Array data attributes consist of three components: An expression denoting the lower bound (if missing in the concrete program, the constant 1 is inserted by the translator), an expression denoting the upper bound and the data attributes of the elements of the array:

![Fig. 2.9 array (data attributes of arrays)](image)

(b) Structures. In a similar way to arrays, structures are described recursively. A structure is analogous to a one-dimensional array, whose elements may have any data attributes: array, structure, cell, 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 data attributes, and a complete structure description has the form:

![Fig. 2.10 struct(data attributes of structures)](image)

The number (n) of successors of a structure is called its order. Note, that the main identifier of a structure occurs as selector in the declaration part, while the successor identifiers are components of the data attributes.

(c) Cells. Cells have the same description as structures; additionally there is a flag (the elementary object T) besides the list of successors, which distinguishes cells from structures:
(d) Scalars. By this recursive description of the structuring of data aggregates, one finally comes down to the data attributes of scalars. These are objects describing the properties of the different types of data: mode, base, scale, precision, and scale factor for arithmetic data; base, an expression denoting the length, and a flag distinguishing fixed and varying length for string data; etc. Note, that the elementary object LABEL occurring in the position of a scalar data attribute denotes a label variable, while occurring in the position of a complete declaration it denotes a label constant.

Example: The concrete declaration

DCL 1 Z (5,N:N+M),
  2 A (N) INIT (0,0),
  2 B CELL,
  3 C BIT (M+1),
  3 D FIXED INIT (1)

is translated into:

```
Fig. 2.11 cell
```

```
```

2.3.1
Note: The LIKE attribute of a concrete program is resolved by the translator and replaced by a copy of the referenced structure successors.

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

(a) The expressions denoting extents (lower and upper bounds of arrays, string length, 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).

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

(c) The data attributes of restricted label variables (i.e. those specifying a restricting label constant list) are replaced by the object LABEL, since the restricting label lists are also irrelevant for storage mapping.

These evaluated data attributes contain exactly the information necessary for storage mapping of the described aggregate.
**Example:** The evaluated data attributes of the declaration in the previous example (Fig. 2.12) have the following structure (assuming that N has the value 2 and M the value 4):

Note that in this picture digits in the boxes denote values, while in the previous ones they denote the corresponding constants.
### 2.3.2 Entry declarations

The declaration of an entry identifier consists of four components:

![Diagram of entry declaration](image)

**Fig. 2.14** entry declaration

These components are:

(a) The scope attribute INT, EXT or PARAM in the same way as for proper variables (and file identifiers).

(b) The parameter descriptor list as specified in the ENTRY attribute in the concrete program. If no parameter descriptor list is specified, this component is the elementary object *, otherwise a list of parameter descriptors (Fig. 2.15):

![Diagram of param-list](image)

**Fig. 2.15** param-list

A parameter descriptor is an incomplete declaration of the corresponding type (proper variable, entry, file) or the elementary object * if no type is specified at all. Saying a parameter descriptor is an incomplete declaration means that the following components are missing: scope attribute, initial set, successor identifiers for proper variable descriptors, scope attribute and return type for entry descriptors, scope attribute, title and environment attribute for file descriptors. All other components are completed by default rules as for declarations if not explicitly specified in the concrete program.

(c) The return type, i.e. either the scalar data attributes specified in the RETURNS attribute in the concrete program or those resulting from default rules.
(d) For internal entry declarations (and the external entry declarations in the
declaration part of the program, cf. 2.1) as denotation an integer value serv-
ing as a pointer into the body list of the block to which the declaration is
local. For external and parameter declarations this component is $0$.

Depending on the three different scope attributes, the correspondence between
an entry declaration and its procedure body is specified in the abstract program
in the following ways:

(a) Internal. For internal entry declarations, the body list component of the
block to which the declaration is internal contains one body, which has an
entry point with the declared identifier. The number of this body in the
list is the denotation component of the entry declaration.

Example: In the example given in 2.2 the declaration of the entry identifier
$P$ contains as denotation component the integer value 1 pointing to the first
(and only) element of the body list.

(b) External. For the external entry declarations in the declaration part of the
program (cf. 2.1) the correspondence to their procedure bodies is given in
the same way as for internal entry declarations.

Example: In the example given in 2.1 each of the declarations of $A$, $B$, $C$
contains the integer value 1 as denotation component, the declaration of $D$
the integer value 2 and the declaration of $C$ the integer value 3; these integers
refer to the three elements of the body list:

![Diagram of body list and entry declarations]

Fig. 2.16
For each external entry declaration in the declaration part of a block (begin block or procedure body, but not the program), the denotation component is $\varnothing$. In this case there exists also an entry declaration of the same entry identifier in the declaration part of the program. The procedure body corresponding to this entry declaration (as described above) corresponds to all external entry declarations of this entry identifier in any block in the program.

(c) Parameter. For entry parameters the correspondence to a procedure body is established during interpretation by passing an entry argument to the parameter.

2.4 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 (assignment statement, allocate statement, null statement, etc.) including the if-statement and on-statement, the different types of do-groups, and the begin block. The term does not include: declarations, procedure bodies, format sentences and incomplete clauses as, e.g. BEGIN; or DO I = 1 TO N; or IF I > 1 THEN or END; etc. So, the term "statement" does not denote a syntactical unit delimited in the concrete program by a semicolon but a logical unit that may appear "in a statement position", e.g. as THEN alternative of an if statement, and that may in some way be executed independently from other program parts.

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

(a) a condition part, representing the condition prefixes of the concrete program,
(b) a label list, namely a list of statement label identifiers,
(c) the proper statement:

```
Fig. 2.17 statement
```
There are 23 different types of proper statements. The structure of most of them is a nearly one-to-one mapping of their syntactical structure in the concrete program (some default constants are inserted by the translator if no expressions are specified in the concrete program, e.g., the constant 1 as by-expression of a do-specification).

2.4.1 Begin block

In the structure of a program, a begin block has the same position as any statement, e.g., an assignment statement. Therefore a begin block is a proper statement like all other proper statements.

On the other hand, a begin block introduces a new level of structuring into a program, since it contains a statement list which may include any type of statement, including again begin blocks.

The structure of a begin block is nearly the same as that of a procedure body (cf. 2.2). The only difference is that the statement list of a begin block does not contain entry points.

The difference between a begin block and a procedure body is that a begin block is an element of a statement list and is to be executed during the sequential execution of this statement list, while a procedure body is declarative information to be activated by a call statement by means of an entry identifier. This difference implies both different occurrence in a program (in a statement list or body list, respectively) and different syntactical structure (existence or not of entry points).
2.4.2 Group and statement list

There are two essentially different types of "do-groups" in a concrete program: those without iteration specification and those with iteration specification.

Those without iteration specification are not called "groups" in the abstract program. They are simply statement lists (or procedure statement lists if occurring in a procedure body), which are parenthesized by DO; ...; END in the concrete program. Since by the structuring of an abstract program such parentheses are not necessary, they occur just as statement lists (or procedure statement lists) in the position of a proper statement.

The do-groups with iteration specification of the concrete program are called groups in the abstract program. Essentially they are translated one-to-one from the concrete to the abstract program, except that all declarative information is extracted into the declaration part and body list of the containing block (the same is valid for the statement lists without iteration specification). Since no transfer of control is allowed from outside into a group, a group (in contrast to a procedure statement list) may not contain entry points.

2.4.3 Allocate statement

The only simple statement of PL/I which has a somewhat different structure in an abstract program from in a concrete program is the allocate statement.

First, the information about one data aggregate to be allocated is composed into an object which is structured similarly to a proper variable declaration (cf. 2.3.1), especially in its data attributes and its initial set.

Second, the successor identifiers of structures are omitted, since they are redundant (they are tested by the translator against the successor identifiers of the corresponding declaration).
Example: The concrete statement

```
ALLOCATE 1 X(N:N+M), 2 A INIT(0), 2 B INIT(1), 1 C BIT(5);
```

is translated into the following object:

```
<table>
<thead>
<tr>
<th>5-St</th>
<th>I I</th>
<th>5-allocate-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLOCATE</td>
<td></td>
<td>elem(1)</td>
</tr>
<tr>
<td>s-id</td>
<td></td>
<td>s-al</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>s-lbd</td>
</tr>
<tr>
<td>{N}</td>
<td></td>
<td>s-ubd</td>
</tr>
<tr>
<td>elem(1)</td>
<td></td>
<td>s-elem</td>
</tr>
<tr>
<td>s-init-set</td>
<td></td>
<td>elem(2)</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>s-base</td>
</tr>
<tr>
<td>BIT</td>
<td></td>
<td>s-length</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Fig. 2.19

Note: All simple statements (i.e., all statements except block, group, statement list) have a component giving the statement type (s-st component). In most cases, this component is redundant, since the structure uniquely determines the statement type. But in some cases, there are pairs of statement types (allocate and free, open and close, get and put, read and write) which, at least in special cases, may not be distinguishable by the structure of the statement alone.
Expressions are decomposed by the translator into (possibly nested) "elementary expressions". Each of them is:

(a) an **infix expression**, consisting of two operands and an infix operator:

```
  s-operator
     \   / \
    /     \ 
operator expression1 expression2
```

Fig. 2.20 infix-expr

or:

(b) a **prefix expression**, consisting of an operand and a prefix operator:

```
  s-operator
     \   / \
    /     \ 
operator expression
```

Fig. 2.21 prefix-expr

or:

(c) a **parenthesized expression**, consisting of an operand only:

```
  s-op
     \   / \
    /     \ 
expression
```

Fig. 2.22 paren-expr

or:

(d) a **reference** (described below, cf. 2.5.1)

or:

(e) a **constant** (described below, cf. 2.5.2)

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 parenthesized expressions 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.

**Example:** The concrete expression

\[-A \ast B + (X + Y) / C\]

is translated into the object:

![Diagram of the translation process]

The final components of an expression are references and constants.

### 2.5.1 References

A reference to a variable occurring in an expression consists of an abstract program of three components:

![Diagram of reference structure]
(a) The pointer qualifier. This component is optional; it is possible only for references to based variables and denotes the pointer used for the reference.

(b) The identifier list. It is the fully qualified name of the referenced variable. That means the following:
If the reference refers to a component of a structure, then the identifier list consists of the main identifier of the complete aggregate, followed by the 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.

(c) The argument list, consisting of expressions and possibly asterisks. Since in a concrete program the subscripts may be added arbitrarily to any identifier of the qualified name, in the abstract program they are collected into a separate list.

Example: The concrete reference
\[ A(1) \cdot B \cdot C (*, M + N) \]
referring to the declaration

\[ \text{DCL } S(5,0:10), 2 A, 3 B(2 \cdot N), 4 X, 5 C(10), \ldots \]
is translated into:

\[ \text{s-id-list} \quad \begin{array}{c}
\text{elem(1)} \\
\text{elem(2)} \\
\text{elem(3)} \\
\text{elem(4)} \\
\text{elem(5)}
\end{array} \quad \text{s-arg-list} \begin{array}{c}
S \\
A \\
B \\
X \\
C \\
1 \\
* \\
M+N
\end{array} \]

Fig. 2.25

Note: References to other identifiers than those of variables (e.g., function references) never contain pointer qualifications and have identifier lists consisting always of one identifier only.
### 2.5.2 Constants

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

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

(b) 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, which is an operand as produced during interpretation when evaluating an expression. This is an object consisting of the scalar data attributes implied from the form of the constant, and the (implementation dependent) value representation of its value.

![Fig. 2.26 const (constant)](image)

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

**Example:** The concrete constant

007.30

is translated into:

![Fig. 2.27](image)

where the value representation vr is \( vr = \text{represent}(da, 7.3) \) and da is the data attribute shown in the figure.
There are two different kinds of constraints on the concrete
representation.

(a) In positions where only (symbolic or constant) integer
conditions are given, the

(b) as part of a constraint.

In the latter case, the function programs continue with the value of the

In the former case, an elementary object enrichment formation is

In the second case, the transformator heuristics are applied, giving us

and the (representation tableau) value interpretation of the name.

Example of the concrete constraint:

is transferred into:

"The data structure can only be interpreted at high levels of detail."

"Prefer the name representation at a low level of abstraction."
3. THE STATE OF THE PL/I MACHINE

A detailed description of the structure of the state of the PL/I machine may be found in [6], chapter 2.

The present chapter will deal with three aspects under which one can consider the immediate components of the state, namely their use, their scope and their structure. There are 18 different immediate components of any state of the PL/I machine to be considered.

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

- **S** Internal Storage
- **ES** External Storage
- **UN** Unique Name Counter
- **DN** Denotation Directory
- **AT** Attribute Directory
- **FU** File Union Directory
- **TD** Time and Date Part
- **ET** Event Trace
- **PA** Parallel Task and Event Part
- **TE** Task and Event Specification
- **AG** Aggregate Directory
- **FD** File Directory
- **E** Environment
- **EI** Epilogue Information
- **CS** Condition Status
- **CI** Control Information
- **D** Dump
- **CI** Control Information

The use of the components:

1. **tasking: PA, TE, ET**

The parallel task and event part **PA** contains for each active task its local state components. Before an instruction of a specific task is executed its local state components are brought to the surface of the current state in the position of the respective immediate components.

The task and event specification **TE** contains all information necessary for proper control and, in particular, for termination of a task.

The event trace **ET** records all completions of event variables and all starts of executions of wait statements.

1) *i.e.* the state components which are only used by the specific task.
(2) block activations: D, EI

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.

(3) interpretation of statement lists: CI, C

The control information CI is a stack which reflects the dynamic nesting of compound statements (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: E, DN, AT

The environment E reflects the scope rules. It contains all identifier lists (i.e. all fully qualified identifiers) which may be referenced in the current block activation.

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, S contains the values of variables.

(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 and may be considered as the counterpart of AG. 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 but never decreased.

(8) **condition status: 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) **time and date part: TD**

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

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** if it belongs to the entire interpretation of a program, **block local** if it belongs to a specific block activation and **task local** if it is a private state component of a specific task.

The following lists the state components according to their scope:

(1) **program local:**

S, ES, UN, DN, AT, FU, TD, M, ET, PA

(2) **task local:**

TE, AG, FD

(3) **block local:**

E, EI, CS, CI, C, D
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 are directories in the above sense:

DN, AT, FU, PA, AG, FD, E.

(2) stacks:

A stack always reflects some parenthesis-structure. The following three components are stacks:

D, CI, C.

D reflects the dynamic nesting of block activations and CI reflects the dynamic nesting of compound statements within a block activation.

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

Among other things C reflects the parenthesis-structure of expressions during their interpretation.
4. STORAGE AND DATA

4.1 Representation of Data in the PL/I Machine

In this chapter the way in which data are represented in the PL/I machine is described. Whereas the primary mathematical concept is that of a value, the forms in which data actually appear in the state components of the machine are 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:

a) A value cannot always be stored and retrieved without loss of accuracy.
b) In case of numeric pictures, the language explicitly defines how a numeric value is to be represented.
c) Value representations permit a natural treatment of overlay defining and of cells.

An operand consists of a data attribute and a value representation. Operands have been introduced because many operations depend not only on the values but also on the data attributes of their arguments.

4.1.1 Values, value representations, operands

There are different types of values; they are associated with different types of scalar data attributes (Fig. 4.1):

<table>
<thead>
<tr>
<th>data attributes</th>
<th>associated types of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic, numeric picture</td>
<td>numerical values</td>
</tr>
<tr>
<td>character string, character picture</td>
<td>character string values</td>
</tr>
<tr>
<td>bit string</td>
<td>bit string values</td>
</tr>
<tr>
<td>LABEL</td>
<td>label denotations, format denotations (cf. 5,9)</td>
</tr>
<tr>
<td>POINTER, OFFSET</td>
<td>pointers (cf. 4.2.1)</td>
</tr>
<tr>
<td>TASK</td>
<td>integer values (cf. 4.1 of /6/)</td>
</tr>
<tr>
<td>EVENT</td>
<td>event values (cf. 4.1 of /6/)</td>
</tr>
</tbody>
</table>

Fig. 4.1 Scalar data attributes and their associated types of values
A numerical value 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 character string value is a list of character values, a bit string value a list of bit values. Examples for character values are the objects A-CHAR, B-CHAR, ..., O-CHAR, I-CHAR, ...; the two bit values are the objects O-BIT and I-BIT. Note that 1, I-CHAR, I-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.

In the internal storage of the PL/I machine, values are represented by certain implementation-defined objects called value representations. Thus, interpretation of a reference to a variable eventually leads to an application of the function el-ref (stg) which yields a value representation, the "contents" of the storage stg; similarly, interpretation of an assignment to a variable eventually to an application of the function el-ass (p, vr, stg) which changes the contents of the part p (stg) of the storage stg to the value representation vr (cf. 4.2.2).

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 data attribute da, and to retrieve v from its representation, da is needed again.

Frequently, not only the values, but also the attributes of data are needed. An operand is an object consisting of two components, a data attribute da and a value representation vr (Fig. 4.2). The result of the evaluation of an expression, and the arguments to many operations (infix operators, prefix operators, most of the builtin functions, conversion) are operands.

The da-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

Let da be an evaluated scalar data attribute of one of the types listed in Fig. 4.1, with no precision or string length that is specified by *. The transition between a value v and its representation vr (with the given attribute da) is illustrated by Fig. 4.3:

![Transition Diagram](image)

Fig. 4.3 Transition between a value and its representation

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

Consider the set v-1-set(da) of the values that are assumed by value(da, vr) if vr ranges over vr-set(da); this is a subset of the set v-set(da). The following postulate is made on the functions represent(da, v) and value(da, vr):

For all elements vr of vr-set(da) and all elements v of v-1-set(da) the two relations

\[ vr = \text{represent}(da, v) \quad \text{and} \quad v = \text{value}(da, vr) \]

are equivalent; i.e., represent(da, v), considered as function over v-1-set(da) only, and value(da, vr) are inverse functions.

In view of this postulate, v-1-set(da) can be called the set of exactly representable values, i.e., of the values for which transition from a value to its representation, and from the representation back to its value, results in the unchanged value. (On the other hand, those values of v-set(da) which are not in v-1-set(da) are certainly not exactly representable, because the function value(da, vr) always leads into v-1-set(da).)
Examples

1. Let $\text{da}$ be the attribute CHAR (4). The set $v-1$-set (da) of exactly representable values is the set of all character strings of length 4, whereas the set $v$-set (da) is the set of all character strings. Hence, the string 'ABCD' will be exactly representable, whereas the strings 'ABC' or 'ABCD' will not; the values of the representations of the latter two strings will be 'ABCb' and 'ABCD', respectively (where b denotes blank) (cf. 4.1.2.1).

2. Let $\text{da}$ be REAL DEC FIX (4,1). The set $v-1$-set (da) is implementation-defined, the set $v$-set (da) is the set of all numerical values that will not raise the SIZE condition. It will be guaranteed that the number 123.4 belongs to $v-1$-set (da), but not, that the number 123.45 belongs to $v-1$-set (da); 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 (cf. section 4.1.2.2).

Note: A generalization of the case treated so far would be to permit not only values that, though having representations, can never be reached by value (da,vr) (Fig. 4.3), but also representations that, though having values, can never be reached by represent (da,v) (Fig. 4.4):

![Fig. 4.4 A generalization of Fig. 4.3](image URL)

In this case, represent (da,v) and value (da,vr), considered only as functions over the subsets $v-1$-set (da) and $v$-set (da), respectively, would have to be postulated as inverse functions. The situation actually arises for floating-point numeric picture da: represent (da,v) always produces a representation with normalized mantissa, but unnormalized representations may be encountered through overlay defining.

Elementary and non-elementary cases.

In the formal definition of the functions represent (da,v) and value (da,vr), certain cases, the "non-elementary" cases, are defined explicitly, in terms of the functions rep (da,v) and val (da,vr) which treat the "elementary" cases; hence...
the fact that (and the extent to which) represent \((da,v)\) and value \((da,vr)\) are inverse functions has to be proved, for the non-elementary cases, from their definitions. The relation between rep \((da,v)\) and val \((da,vr)\) is postulated axiomatically. Furthermore, in some cases, a test has to be made as to whether the value is representable; if it is not, the SIZE or CONVERSION condition will be raised. For these cases, an instruction \texttt{test-rep}(da,v)\ is defined instead of a function represent \((da,v)\). For representable \(v\), test-rep \((da,v)\) behaves like a function in that its only effect is to yield the representation of \(v\); it is this function, whether it is called represent \((da,v)\) or not in \cite{6}, which is meant in Fig. 4.3 and 4.4.

4.1.2.1 The non-elementary cases

1. For complex arithmetic \(da\), the real and the imaginary part of the numerical value \(v\) are represented separately (with a data attribute that differs from \(da\) only in that it has real mode). If the two parts are representable and yield representations \(vr_1\) and \(vr_2\), then the representation of \(v\) is an object composed of \(vr_1\) and \(vr_2\) (Fig. 4.5):

\[
\begin{array}{c}
\text{s-real} \\
vr_1 \quad vr_2 \\
\end{array}
\]

Fig. 4.5 Representation of a complex value

For real arithmetic \(da\), if \(v\) is a complex value, only its real part is represented; for the case of a real value \(v\), see 4.1.2.2.

2. For string \(da\), the representation is a list of the representations of the single characters or bits of the string value \(v\). (To represent single character or bit values, the "degenerate" data attributes CHAR or BIT are used, cf. next section). The set \(v\text{-}1\text{-set}(da)\) is the set of character string values, or bit string values, whose length satisfies the requirement prescribed by \(da\). The set \(v\text{-set}(da)\) is the set of all character string values, or bit string values, and when necessary, \(v\) is transformed into an element of \(v\text{-}1\text{-set}\) by truncation, or by extension with BLANK or 0-BIT.

3. For numeric picture \(da\), the numeric value \(v\) is first transformed into a string value (cf. 8.3.3), which is then represented as above; for character picture \(da\), the character string value \(v\) is represented as above, except that the CONVERSION condition is raised if \(v\) does not match the picture specification.
4.1.2.2 The elementary cases

These are the cases listed in Fig. 4.6 (with the two special attributes CHAR and BIT introduced as explained above):

<table>
<thead>
<tr>
<th>data attributes</th>
<th>associated types of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>real arithmetic</td>
<td>real numerical values</td>
</tr>
<tr>
<td>CHAR</td>
<td>single character values</td>
</tr>
<tr>
<td>BIT</td>
<td>single bit values</td>
</tr>
<tr>
<td>LABEL</td>
<td></td>
</tr>
<tr>
<td>EVENT</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4.6 Data attributes and their associated types of values:

the elementary cases.

For real arithmetic da, let \( b \) be the radix of da, i.e. \( b = 10 \) or \( b = 2 \) according to decimal or binary base of da; let \( p \) be the precision of da, and, for fixed-point da, \( q \) the scale factor of da.

The set \( v\text{-set}(da) \) of representable real numerical values is the set of all real numerical values \( v \) such that

\[
\begin{align*}
\text{a) for fixed-point da:} & \quad |v| b^q < b^p \\
\text{b) for floating-point da:} & \quad v = 0 \text{ or } \minflt_{da} \leq v < \maxflt_{da} \\
\end{align*}
\]

where \( \minflt_{da} \) and \( \maxflt_{da} \) are certain implementation-defined limits (depending only on \( b \)).

For real values outside \( v\text{-set}(da) \), the SIZE condition will be raised.

The set \( v\text{-1-set}(da) \) of exactly representable values is implementation-dependent. It will, however, contain the subset \( v\text{-0-set}(da) \) (cf. Fig. 4.7.1) defined as follows: \( v\text{-0-set}(da) \) is the set of all values \( v \) of \( v\text{-set}(da) \) such that

\[
\begin{align*}
\text{a) for fixed-point da:} & \quad |v| b^q < b^p \\
\text{b) for floating-point da:} & \quad v = 0 \text{ or } \minflt_{da} \leq v < \maxflt_{da} \\
\end{align*}
\]
a) for fixed-point da: \( v.b^q \) is an integer  
b) for floating-point da: \( v = m.b^e \), where \( m \) and \( e \) are integers and \(|m| < b^p\).

![Diagram](image)

**Fig. 4.7.1 Transition between a real value and its representation**

The definition of \( v-0\text{-set}(da) \) 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 \( v\text{-set}(da) \)) have been expressed as limits for the entire value, not for the exponent.

**Example:**

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

\[
0, \pm 0.1, \pm 0.2, \ldots, \pm 999.8, \pm 999.9.
\]

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

For the other elementary attributes da, both \( v-1\text{-set}(da) \) and \( v\text{-set}(da) \) are the set of all values whose type is associated with da, i.e. all these values are exactly representable (Fig. 4.7.2):

![Diagram](image)

**Fig. 4.7.2 Transition between a non-real elementary value and its representation**
4.2 Internal Storage and Generations of Variables

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.

We say that the storage part of the PL/I machine is of type storage. Given an entity of type storage one may define the parts of this entity. There are functions which select parts out of a storage, which are called pointers. Given a storage stg and a pointer p we call p(stg) the p-part of that storage. If stg has no p-part, then p(stg) is undefined and we say that p is not applicable to stg.

A two-dimensional picture may illustrate the relations between a storage stg and its parts:

![Diagram of storage parts](image)

Any two storage parts may or may not be independent. Being independent means that the two parts have no storage in common. Non-independent storage parts are shown in Fig. 4.8 by overlapping regions, i.e., p_1(stg) is independent of p_3(stg), but not independent of p_2(stg).

The following statement holds:

The relation of two storage parts p_1(stg) and p_2(stg) being independent or not is only a property of the two pointers p_1 and p_2.
The characteristic property of a storage or storage part is its **size**. The size determines which pointers are applicable to a storage and, consequently, which parts one may select from it. The following statement holds:

The size of a storage part \( p(\text{stg}) \) is only a function of the pointer \( p \).

The size of a storage fully determines its capacity with respect to storing information, i.e., with respect to keeping value representations and, if used as an area, making allocations in it. The size is an implementation-defined quantity. It is not necessarily simply a numerical value.

**Example:** A linear bit storage may serve as a concrete example. It consists of a linear arrangement of single-bit stores, indexed from 1 up to a maximum index \( n \), \( n \) is the size of the storage. Each pointer to that storage is a function with two integer arguments \( f(i_1, i_2) \). It is applicable to a storage \( \text{stg} \) of size \( n \) if \( 1 \leq i_1 \leq i_2 \leq n \). \( f(i_1, i_2)(\text{stg}) \) denotes the subpart of \( \text{stg} \) between and including the \( i_1 \)th and the \( i_2 \)th element. Two storage parts \( f(i_1', i_2')(\text{stg}) \) and \( f(i_1'', i_2'')(\text{stg}) \) are independent if \( 1 \leq i_1', i_2' \leq n \), \( i_1' \neq i_2'' \), and either \( i_2' < i_1'' \) or \( i_1' < i_2'' \).

\[
\begin{align*}
\text{f}(2,5)(\text{stg}) & \quad \begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array} \\
\text{f}(4,9)(\text{stg}) & \quad \begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array}
\end{align*}
\]

Fig. 4.9 Linear storage model

The \( p_1 \)-part of the \( p_2 \)-part of a storage \( \text{stg} \) is defined by \( p_1(p_2(\text{stg})) \), or \( p_1 \cdot p_2(\text{stg}) \). The symbol \( \cdot \) is used for functional composition. The term \( p_1 \cdot p_2 \) represents again a pointer.

### 4.2.2 Elementary assignment and reference

Each storage \( \text{stg} \) has associated with it a value representation, obtained by the function \( \text{el-rf}(\text{stg}) \). The size of a storage determines the range of value representations that can be associated with it. The value representation associated with the \( p \)-part of \( \text{stg} \) is given by \( \text{el-rf}(p(\text{stg})) \). The value representation associated with the \( p \)-part can be changed by the elementary assignment function \( \text{el-ass}(p, vr, \text{stg}) \), provided that \( vr \) is within the range determined by the size of \( p(\text{stg}) \). This function gives a new storage \( stg' \). \( stg' \) has the same properties as
stg just with the exception that vr is now the value representation associated with the p-part. The value representations associated with those 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 have different value representations associated after the assignment. Since no relationships between value-representations of parts of storage are defined, an assignment simply makes the value-representations of all storage parts unknown which are not independent of the part to which the assignment is made (with the exception of this part itself). 1)

Example: Let the storage parts of the storage in Fig. 4.8 have the following value representations associated:

\[
\begin{align*}
&\text{p}_1 (\text{stg}) \ldots \text{vr}_1 \\
&\text{p}_2 (\text{stg}) \ldots \text{vr}_2 \\
&\text{p}_3 (\text{stg}) \ldots \text{vr}_3
\end{align*}
\]

After execution of the elementary assignment el-ass(\text{p}_1, \text{vr}, \text{stg}) the situation is:

\[
\begin{align*}
&\text{p}_1 (\text{stg}) \ldots \text{vr} \\
&\text{p}_2 (\text{stg}) \ldots \text{unknown} \\
&\text{p}_3 (\text{stg}) \ldots \text{vr}_3
\end{align*}
\]

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 storage part is noted in order to prevent further allocations from using it.

An allocation can be made either in the main storage \(S\), or (for based variables) in an area (which is itself part of the main storage). The main storage and each area have a certain part, the allocation state part, for holding the allocation state. The allocation state is the set of pointers identifying those parts which have been used for allocations. Let \(z\) be the size of an area, then \(\text{allst-part}(z)\) is that part of the area holding the allocation-state. If the \(\text{p}_1\)-part of the area is used for a new allocation, then \(\text{p}_1\) is added to the allocation state.

1) There are exceptions to this in string assignment.
Fig. 4.10 An area is allocated in the p-part of $S$. The $p_1$- and $p_2$-part of the area have been used for allocations. $p$ is noted in the allocation state part of $S$ and $p_1$ and $p_2$ are noted in the allocation state part of the area.

On freeing a variable the pointer identifying the storage part to be released is deleted from the allocation state of the respective storage.

The storage part which is reserved on allocation of a variable is given by an implementation defined function which depends on the data attributes and density (PACKED or ALIGNED) of the variable and on the allocation state of the storage (the main storage or an area) in which the allocation is made. The selected storage part must fulfill the following requirements:

1. Its size must be such that it can hold 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 allocation state part of the storage in which the allocation is made.

If no storage part having the above properties can be identified the allocation is not possible. This situation is called storage-overflow. 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 chapter.
4.2.4 Storage mapping

The properties of a variable which are of interest in connection with storage mapping are the data attributes and the density (the PACKED or ALIGNED attribute).

A variable, according to its data attributes, may be a scalar, an array, a structure, or a cell variable. An immediate sub-part of a variable is identified by an integer value. For array variables, this integer value must be in the range between the lower and upper bound of the array, for structures and cells between 1 and the number of elements called the order of the structure or cell, for fixed length strings between 1 and the length of the string. A sub-part of a variable again is of scalar, array, structure, or cell type. Sub-parts of non-scalar sub-parts are identified in the same way as immediate sub-parts of a variable. A sub-part of a variable therefore is identifiable by a list of integers, which is called the reference list.

An immediate sub-part of a variable is said to be to the left of another immediate sub-part, if the integer identifying it is smaller than the integer identifying the other sub-part. This generalizes in an obvious way to non-immediate sub-parts.

The way the various parts of a variable are associated with storage is determined by the storage mapping function. Let the storage allocated for a non-scalar variable with data attributes da and density dens be \( p(S) \). The storage mapping function \( \text{map}(\text{da}, \text{dens}, i) \) gives a pointer \( p_i \) such that the \( i \)th immediate sub-part of the variable is associated with the storage part \( p_i(p(S)) \). The following requirements must be satisfied:

1. if a variable or the sub-part of a variable is an array or a structure, the storage parts associated with the immediate sub-parts must be mutually independent.
2. for each scalar data attribute there is a certain implementation-defined set of value representations that may be associated with a variable given this attribute (cf. 4.1.2). The size of the storage associated with a scalar variable, or the scalar part of a variable, must be such that it can hold any value representation determined by the data attribute.

It is important that the requirement of independence of immediate sub-parts is not made for cell variables. The consequence is that an assignment to one alternative of the cell variable makes the value representations associated with the other alternatives unknown. If one of these alternatives is an area, then the allocation state of this area is also unknown after the assignment.
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 the storage associated with 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 data attribute part. This part consists of the evaluated data attributes of the variable.
2. The mapping information. This part gives the necessary input to the storage mapping function and consists of evaluated data attributes and density.
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. Subgenerations 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 data attributes \( \text{eda} \) and density \( \text{dens} \), a new generation is formed with:

- **data-attribute part:** \( \text{eda} \)
- **mapping information:** \( \text{eda} \) and \( \text{dens} \)
- **pointer part:** single pointer determined from \( \text{eda} \), \( \text{dens} \) and the allocation state 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 to 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 data-attribute part are equal to the corresponding array bounds and string lengths of the data attributes in the storage mapping part. It follows that a generation formed on allocation of a variable always is connected. Certain PL/I operations (record I/O, overlay defining, application of the ADDR builtin function) are allowed only with variables having connected generations.

4.2.6 Subgenerations of generations

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

1. when an assignment is made to the sub-part (provided that it is scalar)
2. when the operand associated with the sub-part is to be evaluated (provided that it is scalar)
3. when it is passed to the parameter of a procedure (cf 7.5.1).

In a reference to a variable in the program text (cf. 8.2.7) immediate sub-parts of structures and cells are identified by identifiers, immediate sub-parts 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 I X(7, 2) PACKED, 2 Y BIT(3), 2 Z(5) BIT(5);

and a reference to X be

X(1, *) • Z(3+2)

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 data attribute part is formed from sub-data attributes of the data attributes in the data attribute-part of the generation. They are obtained by successively applying the elements of the reference list to determine immediate sub-parts of the data attributes. If the element of the reference list is an integer value $i$, then these sub-parts are:

a) for arrays the data-attributes of the immediate array elements
b) for structures and cells the data attributes of the $i$th structure or cell element.

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

(2) A new mapping information is formed. The data attributes in the mapping information of the generation are treated like those in the data attribute part (see above). The density remains unchanged.

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

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

$$(\text{map}(\text{eda}, \text{dens}, i)) \cdot p$$

where eda is the data attribute of the part of the variable corresponding to $p$. The new pointer part identifies the storage part corresponding to the $i$th immediate sub-part of this variable part.

If the pointer part is a single pointer $p$ and the 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:

$$p_1 = (\text{map}(\text{eda}, \text{dens}, lbd)) \cdot p$$
$$p_2 = (\text{map}(\text{eda}, \text{dens}, lbd+1)) \cdot p$$
$$\vdots$$
$$p_n = (\text{map}(\text{eda}, \text{dens}, ubd)) \cdot p$$
where lbd and ubd are the lower and upper bounds of the array, ubd-lbd = n-1, and eda the array data attribute. The rest of the 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 lead to a non-connected generation.

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

If the pointer part is a list of pointers and the element of the reference list is an asterisk then the rest of the 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 data-attribute part and the pointer part of the sub-generation determined by the reference is illustrated in the following,

Evaluated data attributes eda of \( X \):

![Diagram](image_url)
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-data-attributes defined by the first element of the reference list are:

\[
\text{eda}_1:
\]

\[
\begin{array}{ccc}
\text{s-lbd} & \text{s-ubd} & \text{s-elem} \\
1 & 2 & \\
\end{array}
\]

Fig. 11b

With \(p_1 = \text{map(eda, PACKED, 1)}\) we get the modified pointer part: \(p_1 \cdot p\)

\(p_1 \cdot p(5)\) now is the storage associated with the first element of the array variable X.

(2) The rest of the reference list is \(<*, 2, 5>\). We now have to create a list of pointers, each element corresponding to an element of the array \(\text{eda}_1\).

With \(p_{21} = \text{map(eda, PACKED, 1)}\)

\(p_{22} = \text{map(eda, PACKED, 2)}\)

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

(3) The rest of the reference list is \(<2, 5>\). The sub-data-attributes defined by the first element are \(\text{eda}_2\):

\[
\begin{array}{ccc}
\text{s-lbd} & \text{s-ubd} & \text{s-elem} \\
1 & 2 & \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{s-lbd} & \text{s-ubd} & \text{s-elem} \\
1 & 5 & \text{BIT(5)} \\
\end{array}
\]

Fig. 11c

4.2.6
We now have to modify each element of the above list of pointers. With

\[ P_3 = \text{map}(\text{eda}_{31}, \text{PACKED}, 2) \]

\( \text{eda}_{31} \) being the data-attributes corresponding to each element of the pointer list:

\( \text{eda}_{31} : \)

![Diagram of eda_{31}](image)

we get the modified pointer list <p_3 * p_21 * p_1 * p, p_3 * p_22 * p_1 * p>.

(4) The rest of the reference list is <5>. The sub-data-attributes defined by it are eda_{4}:

\[ P_4 = \text{map}(\text{eda}_{41}, \text{PACKED}, 5) \]

\( \text{eda}_{41} \) being the data attributes corresponding to each element of the pointer list

\( \text{eda}_{41} : \)

![Diagram of eda_{41}](image)
we get the modified pointer list \( <p_4 \cdot p_3 \cdot p_{21} \cdot p_1 \cdot p, p_4 \cdot p_3 \cdot p_{22} \cdot p_1 \cdot p> \).

The resulting sub-generation is composed of the data-attribute part \( eda_4 \), the mapping information consisting of \( eda_4 \) and PACKED, and the pointer part consisting of the above pointer list.

Suppose we would use the reference \( X(1, \cdot) .Z(S) \) as argument to a procedure, where the corresponding parameter \( P \) has the data attributes \( eda_4 \) and is also PACKED. 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 is formed. This sub-generation consists of data-attribute part \( [BIT(S)] \), mapping information \( [BIT(S)] \) and PACKED, pointer part \( p_4 \cdot p_3 \cdot p_{22} \cdot p_1 \cdot p \) (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 data attribute \( BIT(S) \) and the value representation \( el-\text{rf}(p_4 \cdot p_3 \cdot p_{22} \cdot p_1 \cdot p(S)) \).

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 with a storage part, whose size depends on the declared size in an implementation-defined way. A certain part of this storage, the allocation state part, 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. 8.1.1.3). Assignments to area variables are described in 8.2.4.
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 8.2.7.3, the use of pointer variables for allocating and freeing via based variables in 8.1.1.

If p(S) is the storage associated with an area, and o is an offset value identifying a part of the area, then o • 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 conversion between pointers and offsets.

It is important to note that an offset value identifying the storage associated with a variable allocated in an area, only depends on the data attributes and density 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. Similarly, 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 that 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 PACKED and ALIGNED attributes

Variables may be declared with the attribute PACKED or ALIGNED. These attributes serve as argument to the storage mapping function (cf. 4.2.4). The intention of packed 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 packed string aggregates. The location of the various parts in storage is "structure-independent", i.e., the pointer identifying a sub-part depends only on the number of elements (bits or characters) in the sub-part, and on the number of elements (bits or characters) which are to the left of the sub-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, packed string aggregates. The property is significant for the definition of string overlay defining (cf. 8.2.7.2.3).

4.2.7.4 Cells

A variable, or part of a variable, declared as a cell is treated like a structure, with the exception that the alternatives of the cell are not given independent storage on allocation. The alternatives of a cell are treated exactly like elements of a structure. The consequence is that an assignment to one alternative, or part of an alternative, of a cell invalidates the value representations associated with the other alternatives.

There are no data operations defined for cells. Nor do cells expand when written as operands of aggregate expressions.
4.3 External storage and files

The following abbreviations are used in this subchapter:

- at: attribute directory
- bup: bufferpointer
- char: character
- col: column
- csa: complete set of
- descr: dataset description
- ds: dataset
- endf: endfile
- env: environment
- es: external storage
- ev: event, I/O-event
- fd: file directory
- fu: file union directory
- ids: inner dataset
- io: input (and) output
- lsz: linesize
- ods: outer dataset
- ons: onsource
- opt: option
- orig: origin
- pos: position
- psz: pagesize
- ten: task (and) event name
- tmt: transmit
- upd: update

The external storage part ES of the PL/I machine and the access to ES by a file is described in this chapter; this entails the description of the file union directory FU and of the file directory FD.

4.3.1 Organization of external storage

External storage is the repository for datasets, and since any dataset must be available across task and block activation boundaries, the external storage ES is a global state component of the PL/I machine. Every dataset must have two major components, one being the dataset proper, the other serving for identification purposes. The proper dataset again has an optional header label, an optional trailer label, and a data component.

The identification component is more than a selector to retrieve the data- set in ES. This component has to be a collection of datatest names (called dataset titles), every name providing some additional information on the dataset (called a dataset description). There should be no ordering defined over the titles and descriptions of a dataset and over the datasets in ES.
External storage as shown in Fig. 4.12 is an object which has exactly the properties required. With this model as a basis it is useful to refine the terms used in connection with ES.

**titled dataset**  
A member of ES

**dataset title**  
Any of the titles which are part of a titled dataset (e.g., title₁ in Fig. 4.12)

**dataset description**  
Any of the descriptions which are part of a titled dataset (e.g., descr₁ in Fig. 4.12)

**dataset label**  
The header or trailer labels of a titled dataset (e.g., char-list₁, char-list₂ in Fig. 4.12)

**outer dataset**  
The data part of a titled dataset (e.g., outer-dataset₁ in Fig. 4.12)

The representation of dataset titles, dataset descriptions and outer datasets is left to an implementation. The component of each titled dataset selected by s-title-descr-set serves for identification only and thus never provides or receives data, i.e. it remains unchanged throughout the computation. Dataset labels are optional; a missing dataset label is modelled by the empty list <>.
4.3.2 Access to titled datasets

Access to a titled dataset in ES by a title occurs during data transmission to or from an outer dataset or during processing of dataset labels in the case of explicit opening or closing.

Given a certain title, say title₀ (in the form of a list of characters), the titled dataset accessed by title₀ is exactly that titled dataset, say ds₀, which has among its dataset titles a title, say title₁, which matches title₀, i.e. for which the predicate

\[ \text{is-title-match}(\text{title}_0, \text{title}_1) \]

is satisfied. The predicate again is assumed to be implementation defined.

If the predicate is never satisfied, or is satisfied by more than one title of a titled dataset, the access is unsuccessful, and to indicate this the access will yield the null object \( \Omega \). It should be noted that it makes no difference for an unsuccessful access whether the predicate is satisfied for more than one title of one or several titled datasets.

An unsuccessful access in any case is a defined action: the null object \( \Omega \) will be interpreted during opening to execute specific actions, e.g. to raise the undefinedfile condition.

A successful access to a titled dataset in ES by title₀ is uniquely associated with a dataset description, a header label, a trailer label, and an outer dataset. Whenever, later on, the terms title or description (descr) of a titled dataset are used, these terms assume successful access by a given title as a prerequisite.

4.3.3 Dataset mapping

Though a title is associated with an outer dataset if the access is successful, the title contributes nothing to the type of the access. The type of an access depends on a complete set of file attributes (csa) and an environment attribute (io-env), the csa being defined by the language, the io-env being implementation defined.

The type of an access together with the description of a titled dataset are the only parameters to a mapping function "decipher" which, when applied to
its arguments, defines a mapping of the accessed outer dataset (ods) into an inner dataset (ids):

\[ \text{decipher}(\text{csa}, \text{io-env}, \text{descr})(\text{ods}) = \text{ids} \]

The domain of the function decipher is defined by:

- \( \text{is-csa}(\text{csa}) \), i.e., csa must be one of 23 complete sets of file attributes
- \( \text{is-io-opt}(\text{io-env}) \}
- \( \text{is-ds-descr}(\text{descr}) \}

The predicates are implementation defined

The range of mappings of outer datasets is defined by the predicate \( \text{is-inner-dataset}(\text{ids}) \). In particular, for a special complete set of file attributes \( \text{csa}_1 \), an inner dataset \( \text{ids}_1 \) from the range of the mapping, is said to be a proper (inner) dataset if

\[ \text{is-proper-dataset}(\text{csa}_1)(\text{ids}_1) \]

is satisfied. The following table enumerates all proper datasets:

<table>
<thead>
<tr>
<th>Number of csa's</th>
<th>If the csa consists of the following members:</th>
<th>the proper dataset is a:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>STREAM, INPUT or OUTPUT</td>
<td>list of characters including extralingual characters, a LINE-DELEMITER and a FILEMARK.</td>
</tr>
<tr>
<td>1</td>
<td>STREAM, OUTPUT, PRINT</td>
<td>a list as above, also including a PAGE-DELEMITER, CARR-RETURN and TABULATOR.</td>
</tr>
<tr>
<td>8</td>
<td>RECORD, SEQUENTIAL, KEYED, etc.</td>
<td>list of keyed record data including FILEMARK</td>
</tr>
<tr>
<td>8</td>
<td>RECORD, SEQUENTIAL, etc. but not including KEYED</td>
<td>list of unkeyed record data including FILEMARK</td>
</tr>
<tr>
<td>4</td>
<td>RECORD, DIRECT, KEYED, etc.</td>
<td>set of keyed record data</td>
</tr>
</tbody>
</table>
In other words, if the csa contains the attribute STREAM a proper dataset is a list of stream data (where FILEMARK is also termed data); if the csa contains the attribute RECORD a proper dataset either is a list or set of record data (keyed or unkeyed, with or without FILEMARK).

Stream data are elementary objects; record data are objects having an optional key which is a list of characters, a size, and a value representation (cf. Chapter 4.2.2 and 9).

The above table of csa's and proper datasets is a definition of the predicate is-proper-dataset(csa). It is also a definition of the predicate is-inner-dataset(ids) since it is sufficient that the set of inner datasets is the union of all proper datasets.

It is now possible to define the term dataset as follows:

$$\text{dataset} \quad \text{an outer dataset or an inner dataset}$$

Whenever later on the notion of a dataset will be used it will be clear from the context whether an outer or inner dataset is meant.

4.3.3.1 Data transmission from external storage

Data transmission from an outer dataset will turn out to be transmission by an input or update file; this means that one of the attributes INPUT or UPDATE must be a member of csa.

Let the access to the outer dataset ods be characterized by csa, io-env, and descr, then

$$\text{is-proper-dataset(csa)}(\text{decipher(csa,io-env,descr)}(ods))$$

is either true or false, and if it is true there is an inner dataset from which data transmission may in fact take place.
4.3.3.2 Data transmission to external storage

Data transmission to an outer dataset will turn out to be transmission by an output or update file, having the attribute OUTPUT or UPDATE in csa.

Let the access to the outer dataset ods be characterized by csa, io-env and descr and let ids be an inner dataset which satisfies the predicate is-proper-dataset (csa) (ids) then

\[
\text{cipher}(\text{csa}, \text{io-env}, \text{descr})(\text{ids}) = \text{ods}
\]

This means that under the above conditions the function cipher applied to its arguments is the inverse of the function decipher applied to the same arguments. Put in another way, the transition between an inner and an outer dataset may take place in both directions. Thus an outer dataset can appear as an inner, and vice versa. However, one and the same outer dataset can have an arbitrary number of appearances as an inner dataset.

4.3.3.3 Some small examples

Example 1: Let ods_1 be an outer dataset being accessed in such a way that the description of the titled dataset descr_1 describes the outer dataset as a "keyed" one, that the environment attribute io-env_1 specifies "no blocking" and that the complete set of file attributes csa_1 is \{RECORD, UPDATE, SEQUENTIAL, BUFFERED\}.

In this case the implementation would be unable to map ods_1 into the proper dataset because descr_1 and csa_1 conflict, i.e.

\[
\text{is-proper-dataset}(\text{csa}_1)(\text{decipher}(\text{csa}_1, \text{io-env}_1, \text{descr}_1)(\text{ods}_1)) = \text{F}.
\]

Example 2: Same as example 1, but instead of csa_1 let csa_2 be the complete set of file attributes, where csa_2 is \{RECORD, UPDATE, SEQUENTIAL, BUFFERED, KEYED\}.

In this case

\[
\text{is-proper-dataset}(\text{csa}_2)(\text{decipher}(\text{csa}_2, \text{io-env}_1, \text{descr}_1)(\text{ods}_1))
\]

is satisfied and the resulting inner dataset ids_2 results from

\[
\text{ids}_2 = \text{decipher}(\text{csa}_2, \text{io-env}_1, \text{descr}_1)(\text{ods}_1).
\]
Because of \( \text{csa}_2 \), \( \text{ids}_2 \) is a list and one could ask for the length of the list, in other words for the number of record data in the dataset. Let this number be \( \text{num}_2 \).

Example 3: Same as example 2; assume that the access is due to the execution of a rewrite statement which changes the \( k \)-th record data (where \( 1 \leq k \leq \text{num}_2 \)).

The result of such a rewrite statement is that \( \text{ids}_2 \) is changed into \( \text{ids}_3 \) (which differ in the \( k \)-th element), then

\[
\text{cipher}(\text{csa}_2, \text{io-env}_1, \text{descr}_1)(\text{ids}_3) = \text{ods}_2
\]

and \( \text{ods}_2 \) replaces the accessed outer dataset in \( \mathcal{E} \).

Example 4: Same as example 2, but instead of \( \text{io-env}_1 \), let \( \text{io-env}_2 \) be the environment attribute which specifies a "blocking factor \( m \)" where \( m > 1 \).

Similarly to example 2, the resulting inner dataset \( \text{ids}_4 \) is

\[
\text{decipher}(\text{csa}_2, \text{io-env}_2, \text{descr}_1)(\text{ids}_4) = \text{ods}_1
\]

and the number of record data is \( m \cdot \text{num}_2 \).

Example 5: Same as example 4, assume that the access is due to the execution of a rewrite statement which changes the \( k \)-th record data (where \( 1 \leq k \leq \text{num}_2 \)).

The result of such a rewrite statement is that \( \text{ids}_4 \) is changed into \( \text{ids}_5 \) (which differ in the \( k \)-th element), then

\[
\text{cipher}(\text{csa}_2, \text{io-env}_2, \text{descr}_1)(\text{ids}_5) = \text{ods}_3
\]

and \( \text{ods}_3 \) replaces the accessed outer dataset in \( \mathcal{E} \).

### 4.3.4 Files and datasets

The previous subchapters have described the access to an outer dataset in \( \mathcal{E} \) by a given title; the notions of a type of an access and of outer and inner dataset have been introduced.

The topic of this subchapter is:

1) To define the type of an access more precisely by giving all other parameters
on which the access may depend (these will be called the **characteristics of the access**),

(2) to define the access completely, i.e. from a file identifier being a component of a piece of abstract program text, down to the dataset.

### 4.3.4.1 The notion of a file

A file includes the access to a titled dataset in $\mathcal{ES}$, its characteristics, and the titled dataset itself.

The access is modelled by a chain beginning with a file identifier (say $\text{id}_1$) which is associated through the environment $\mathcal{E}$ with a unique name (say $\text{n}_1$) which again is associated through the denotation directory $\mathcal{DN}$ with a file name (say $\text{f}_1$); through $\text{n}_1$ an entry in the attribute directory $\mathcal{AT}$ is also accessible but this entry is only of interest as long as the file does not exist. Continuing with the file name $\text{f}_1$ an entry of the file directory $\mathcal{FD}$ is accessible (say $\text{fd-element}_1$) which in any case contains a file union name (say $\text{u}_1$) which again accesses an entry in the file union directory $\mathcal{FU}$ (say $\text{fu-element}_1$); this entry of $\mathcal{FU}$ contains among other components a title which completes the access to the titled dataset in $\mathcal{ES}$.

The subsequent subchapters on $\mathcal{FD}$ and $\mathcal{FU}$ will provide enough detail on their entries; from there it will follow that the characteristics of an access are placed in the $\text{fd-element}$ (in $\mathcal{FD}$), $\text{fu-element}$ (in $\mathcal{FU}$), and the description of the dataset in $\mathcal{ES}$.

As mentioned in chapter 4.3.2 for the title and description of a dataset, the terms $\text{fd-element}$, $\text{fu-element}$ (and some other terms with the same prefixes) have the existence of a file as prerequisite without mentioning this premise in each case.

The existence of a file is bound to opening and closing (cf. chapters 9.1 and 9.4); before opening only the access from the file identifier to the file name in $\mathcal{DN}$ exists; opening usually causes the $\text{fd-element}$ and the $\text{fu-element}$ to be created; both elements will usually be deleted when the file is closed.
4.3.4.2 Organization of the file union directory

The entry of the file union directory FU has the titled dataset and the entry of the file directory FD as interfaces. Since the whole subchapter on external storage and files proceeds in the direction from ES to the file identifier (which is the reverse direction of the access by the file), the FU is described before the FD.

The FU gathers in its fu-elements all information that neither must be in ES, nor needs by necessity be local to a task.

![Diagram of the file union directory FU with one entry shown in more detail](image)

The fu-element for a file as given in Fig. 4.13 consists of two parts, the first of which exists for every file and is constant throughout the existence of the file. This part consists of the complete set of file attributes (fu-csa) valid for the file, the title (fu-at-title) and the environment attribute (fu-io-env) as copied from the entry of the file name (fu-f) which is either the denotation of the file identifier and taken from the entry of ES or is a special file name s-stand-print, and the title (fu-title) which serves to access the titled dataset in ES and is inserted on opening. The copies of the three components fu-at-title, fu-io-env and fu-f are necessary because in some cases the access to a dataset by a file does not start with a file identifier but with a file union name.

The second part of every fu-element contains data transmission characteristics which are shown in Fig. 4.14, and are again composed of two parts.

1) Note that any file has exactly one fu-element but not all entries of FU are associated with a file. This is because there are also fu-elements for the string sources and string targets of get or put statements (see chapter 9.3). For the scope of chapter 4.3 this fact can be ignored.
The first part has three components fu-tmt, fu-endf, and fu-pos which are all optional and subject to changes. In the order mentioned, the components designate a transmission error \(^1\), the endfile status, and the position. The endfile status T occurs only for input or update files (i.e., files having a fu-csa which contains the attributes INPUT or UPDATE); the position is an integer only for stream or sequential files.

The second part, the stream or record transmission characteristics, is shown for record files in Fig. 4.15 and for stream files in Fig. 4.16.

Fig. 4.15 Record transmission characteristics of an fu-element

In Fig. 4.15, the entries fu-bup and fu-key are only made for a buffered file (fu-key only for a buffered keyed file). They denote the pointer to the buffer in storage S and its key, respectively. The component fu-ev is a set containing the unique names of all I/O-events attached by the file (i.e., for which entries were made into the parallel task and I/O-event part PA). The component fu-read is used in a delete statement, and indicates for sequential files whether or not a read statement has been executed previously. The last part called locking information consists of entries which are sets of character lists (i.e., sets of keys) selected by the unique task names of the tasks which share the exclusive file, and which have locked the keys being associated with.

\(^1\) cf. chapter 4.3.5
In Fig. 4.16, the components fu-lsiz and fu-psz remain constant throughout the existence of the file. Both components are inserted on opening, the first being available for an output file, the second for a print file only. Also the current line (fu-line) \(^1\) is available for print files only. The current column (fu-col), the count (fu-count) \(^1\) and the current format list (fu-fol) are available for any (stream) file; the comma status (fu-comma) and the onsource string (fu-ons) are available for an input file only.

The last three components are significant only during the execution of specific statements accessing the file, i.e. fu-fol during an edit-directed get or put statement; fu-comma during a list-directed and fu-ons during a list- or data-directed get statement.

**4.3.4.3 Organization of the file directory**

The file directory FD contains all information on files which must be local to a task. This directory has DN and FU as its interfaces.

---

\(^1\) this component is available through a built-in function
The component \textit{fd-orig} of the \textit{fd-element} shown in Fig. 4.17 indicates whether the file was created in the task to which the \textit{FD} is local (\textit{OWN}), or whether the file was passed to the task (\textit{INH}). The component \textit{fd-fu} is usually the file union name which is the link to the \textit{fu-element}.

4.3.5 Basic data transmission to and from a file

There are two instructions \texttt{tk-dataset} and \texttt{upd-dataset}: one under the assumption that there is a file with the file union name \texttt{u} which takes the outer data-set from \texttt{ES} and returns it as a proper (inner) data-set; the other which transmits a proper data-set \texttt{ids} into \texttt{ES}, thus updating (replacing) the outer data-set. During the execution of these instructions the component \texttt{fu-tmt} of the \textit{fu-element} will be set to the truth value \texttt{T} or to the null object \texttt{Ω} indicating a transmission error or no transmission error. Since the key used to access special record data can have an influence on a transmission error, the key (or in the cases where there is no key the null object \texttt{Ω}) is an additional parameter in the instructions

\texttt{tk-dataset (u,key)} and
\texttt{upd-dataset (u,key,ids)}.

1) the only exception is for the standard system print file (cf. chapter 9.1.1)
4.3.6 Access to a titled dataset by different files

4.3.6.1 Parallel or shared access

Only direct files may share a titled dataset in one or more tasks. Since the definition of the computation resolves any parallel actions to sequential actions (which are executed in unspecified order) the parallel or shared access is in no way different from successive access.

4.3.6.2 Successive access

The meaning of successive access by different files to one and the same dataset is resolved by making statements over the mappings decipher and cipher for all the cases where successive should have a defined meaning.

The problem of re-input of a created dataset can serve as an example, i.e.

- the access of a dataset by an input or update file if the dataset was created by an output or update file (but not by a print file)
- or

the access of a dataset by a direct input or direct update file if the dataset was created by a sequential keyed output or sequential keyed update file

and the dataset on re-input should be recognized to be the same as the created one.
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 an identifier (or identifier list for the case of a qualified reference). 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 components of the information associated with it in common with another name (sharing patterns). Only single tasks will be considered in this chapter except 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 D of the state of the PL/I machine then this is indicated by the diagram:

\[ D \xrightarrow{A} B \]

Fig. 5.1

In other words A is associated with B in D.

The specific constructions by which the associations are realized in the state of the PL/I machine will, however, be suppressed. The above picture (Fig. 5.1) therefore only states that it is possible to retrieve B given A from D in some way which is not further specified.

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.

As an example a state is considered and an identifier which is known at this point of time. According to the definition of the PL/I machine a unique
name \( n \) is associated with the identifier in the current environment \( E \) and in turn among other things this unique name \( n \) is associated in the attribute directory \( AT \) with a composite object which consists of an attribute \( \text{attr} \) and an environment \( \text{env} \). A more detailed description would have to show that one has first to apply a special function to essentially the identifier which yields a selector which when applied to the environment \( E \) yields the associated unique name. This is, however, suppressed and the diagram will simply show the essential linkage (Fig. 5.2).

![Diagram](image)

Fig. 5.2

5.1 Declarations and References

A block has a component which is a declaration part. A declaration part is a collection of declarations.

The interpretation of a block starts (after some preliminary arrangements in the state) with the interpretation of the declaration part of the block. Each time this process takes place a new entity is created for each individual declaration. The term entity in this context means a collection of state components, linked together in some way or another. The components either exist already or are created when the entity is created.

As it will turn out, different 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.

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.

A declaration specifies which kind of entity is to be created upon its interpretation (e.g. variable, procedure, ... ) and how one can refer to the entity and its components (major identifier, substructure names, array bounds, ... ). The simplest way of reference to an entity is by a simple identifier, the most general way to refer to an entity (basic reference) is by a composite object which consists of an identifier list and an argument list (Fig. 5.3).

5.1
The identifier id₁ is called the main identifier. The rest of the identifier list contains the qualifying substructure names or the names of the alternatives of cells ¹). If the referenced entity is not a structure or cell then the identifier list is the one element-list of the main identifier. The argument list represents the subscript list in the case of an array reference, or in the case of a procedure reference the argument list.

It is a property of PL/I that it is the identifier list of a basic reference which determines uniquely the referenced entity. In particular, the main identifier of a basic reference alone is in general not sufficient to determine the referenced entity and on the other hand neither the number of elements in the argument list nor the kind of arguments is used to determine the referenced entity ²).

In any state of the PL/I machine there is a component which associates identifier lists with unique names, called the environment E (Fig. 5.4). At any point of time this component contains all identifier lists which can possibly be used (in this state) to refer to an entity, and the associated unique name is the name of the entity referred to. The entity itself is linked with its unique name.

Fig. 5.4

¹) The list is completed by the translator if the concrete program had only specified it partially, i.e. in the abstract program only fully qualified references can appear.

²) The subscript list in the case of an array reference is of course used to determine a specific element of an array; not, however, to determine the entire referenced array itself. On the other hand the qualifying identifier list of a structure reference is not only used to determine a specific element of a structure but also the entire structure itself.
Let $t$ be a component of a program which contains references to entities which are not declared within that component, i.e. non-local references. A given environment $env$ determines the interpretation of that piece of program, $t$, if it contains the identifier lists of the non-local references, since the entities referred to by the local references are created during interpretation of the piece of program and the entities referred to by the non-local references are determined by the assumed environment $env$. One may say that a pair (Fig. 5.5) consisting of a piece of program and an environment resolving the non-local references forms a unit with a determined interpretation.

$$(t, env)$$

**Fig. 5.5**

unit with determined interpretation

Usually the interpretation of a text $t$ is determined by the environment component of the state in which the text is interpreted. In some instances, however, the interpretation of a piece of program is determined at some state of the machine with the environment of that state, but the actual interpretation is delayed until some later state (which may possibly possess a different environment component).

The following are examples of such situations. The interpretation of a procedure is determined at the state where its declaration is interpreted, but it is done upon call. The interpretation of the attributes of a controlled variable is also determined at the state where the declaration is interpreted. The interpretation (if any) of the attributes is, however, actually performed at the time of allocation of the controlled variable.

The mechanism in the PL/I machine for dealing with environments determine precisely for any piece of program which is interpreted, which references are allowed and to which entities they refer. Clearly if a piece of program is interpreted, this interpretation must be determined by some environment. The following three recursive rules summarize the determination of interpretation of an entire program by associating environments with a program and its components during interpretation (according to the scope rules).
The interpretation of a program is determined by the empty environment\(^1\).

The interpretation of the immediate components of a block is determined by the environment which determines the interpretation of the block updated with the identifier lists, and their associated unique names, of the corresponding declarations of the declaration part of the block.

The interpretation of immediate components of program texts other than blocks is determined by the environment which determines the text.

Updating an environment means adding the associations of the identifier lists corresponding to the declarations and their unique names to the environment and thereby overwriting old identifier lists and unique names in case of conflict.

The following statements are consequences of these rules:

1. Any particular identifier list if contained in an environment can only be associated with a single unique name.

2. Two different identifier lists, starting with the same main identifier, may be associated with the same unique name (Fig. 5.6). The origin of these associations is then and only then the same interpretation of a particular declaration (structure declaration or cell declaration). The two identifier lists then share all information which is linked to the name \(n\).

\[
\begin{align*}
\langle \text{id}_1, \text{id}_2, \ldots, \text{id}_e \rangle & \quad \overleftarrow{E} \quad n \\
\langle \text{id}_1, \text{id}_2, \ldots, \text{id}_m \rangle & \quad \overleftarrow{E} \quad n
\end{align*}
\]

Fig. 5.6

3. Two different identifier lists starting with the same identifier may be associated with different unique names in the same environment (Fig. 5.7)\(^2\).

---

1) Which is the null object for the PL/I machine.

2) This is the reason why the level one identifier is not sufficient to determine the referenced entity. The situation occurs when a level-1 identifier is re-declared with different substructure names.
(4) For any reference one can determine the declaration which will be the origin of all entities which may possibly be associated with the references from the text of the program alone without execution of the program for given data.

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. 5.8) is therefore valid independently of the kind of its declaration.

\[
\begin{align*}
<\text{id}_1, \text{id}_2, \ldots, \text{id}_n> & \quad E \rightarrow \text{n}_1 \\
<\text{id}_1, \text{id}_m> & \quad E \rightarrow \text{n}_2
\end{align*}
\]

Fig. 5.7

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

The attributes, attr, are essentially the attributes from the declaration which created the entity. The environment, env, determines the interpretation of attr (which may contain expressions)\(^1\). The environment is according to the general rule (see 5.1 rule 3.) the environment which determined the entire declaration part that contained the declaration.

Entities created during the same interpretation of a specific declaration part therefore have the same environment (Fig. 5.9).

---

\(^1\) File parameters, which receive their attributes from the corresponding argument are an exception to this rule.
Entities which correspond to different interpretations of the same declaration have the same attributes but not the same environment.

5.3 Proper Variables

In this and the following sections the initial part of the general diagrams leading from an identifier list to an unique name is suppressed.

The general diagram of a proper variable is:

$$
\begin{array}{c}
\text{id-list} \\
\downarrow \text{den} \\
\text{n} \\
\downarrow \text{(attr, env)} \\
\text{id-list'} \\
\downarrow \text{den'} \\
\text{n'} \\
\downarrow \text{(attr, env')} \\
\end{array}
$$

Fig. 5.9

$$
\begin{array}{c}
\text{Fig. 5.10 a proper variables}^{1)}
\end{array}
$$

$$
\begin{array}{c}
\text{gen}_1 = (\text{eda}_j, \text{mi}_j, \text{pp}_j)
\end{array}
$$

$$
\begin{array}{c}
\text{Fig. 5.10 b generations of variables}^{1)}
\end{array}
$$

---

1) The key to the new abbreviations used in the diagram is:

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Definition</th>
<th>Abbr.</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen</td>
<td>generation</td>
<td>vr</td>
<td>value representation</td>
</tr>
<tr>
<td>eda</td>
<td>evaluated data attributes</td>
<td>pp</td>
<td>pointer part</td>
</tr>
<tr>
<td>mi</td>
<td>mapping information</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.3
The Figures 5.10 a and 5.10 b 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. To allocate a variable means, with respect to Fig. 5.10 1), to create a generation \( (\text{gen}_{k+1}) \) and to add the generation as the head of the generation list in \( \text{AG} \).

   The new generation list is then:

   \[
   \langle \text{gen}_{k+1}, \text{gen}_k, \text{gen}_{k-1}, \ldots, \text{gen}_1 \rangle
   \]

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

2. To free a variable means, with respect to Fig. 5.10 1), to delete the head from the generation list. The new generation list is then:

   \[
   \langle \text{gen}_{k-1}, \ldots, \text{gen}_1 \rangle
   \]

3. The current generation is the head of the generation list \( (\text{gen}_k) \).

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

5. Any attempt to free \( \text{gen}_1 \) is an error 2).

6. eda of any generation is produced upon allocation.

---

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.

2) This is part of a mechanism which guarantees that no task frees variables allocated by its mother. Another part of this mechanism, as shown later, is to make \( \text{gen}_1 \) the null generation NULL upon creation of the variable under certain circumstances.
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 $\langle \text{NULL} \rangle$, where NULL is the null generation, and updated by the execution of explicit allocate and free statements.

c) eda is either taken from the previous generation or evaluated 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 $\langle \text{gen}, \text{NULL} \rangle$ 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 $\langle \text{gen}, \text{NULL} \rangle$ 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.
(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.
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.
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

The general diagram for based and defined variables is:

![Diagram](image)

Fig. 5.13

1. **Based variables** (cf. 8.2.7.3)

   Based variables can only have constant extents. Therefore eda could be derived at any point of time from the original attributes attr and it is only an organisational detail of the PL/I machine to derive eda when the declaration is interpreted.

   The env attached to the attributes is, however, not redundant since attr may contain expressions for initialisation to be evaluated upon allocation. Upon reference a generation is temporarily created from eda and the pointer given by the pointer qualification of the reference.

2. **Defined variables** (cf. 8.2.7.2)

   The denotation eda is not redundant since its production may involve expression evaluation. The attributes are evaluated upon interpretation of the declaration. The environment env attached to the attributes is also non-trivial since the attributes, attr, contain the base which is evaluated upon reference.

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

5.5 Files

The general diagram for a file is 1):

First some general rules are given concerning the point in execution when the various components of the above diagram are created. This will be followed by some rules for parameter passing and finally some sharing patterns will be considered.

General rules:

1) The file name f is created during the prepass and is unique for each declaration of internal files and for all declarations of the same file identifier in case of external files. The file names are substituted into the respective declarations during the prepass.

1) The key to the abbreviations is:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csa</td>
<td>complete set of file attributes</td>
</tr>
<tr>
<td>ds</td>
<td>dataset</td>
</tr>
<tr>
<td>ES</td>
<td>external storage</td>
</tr>
<tr>
<td>f</td>
<td>file name</td>
</tr>
<tr>
<td>FD</td>
<td>file directory</td>
</tr>
<tr>
<td>FU</td>
<td>file union directory</td>
</tr>
<tr>
<td>orig</td>
<td>origin (either own or inherited)</td>
</tr>
<tr>
<td>u</td>
<td>file union name</td>
</tr>
</tbody>
</table>
(2) To open a file (either implicitly or explicitly) means to create the structure to the right of the dotted line. An attempt to open a file which has already been opened and not closed changes nothing in the structure shown by the diagram.

(3) To close a file means to delete the entry in FD (i.e., the association \(f_{FD}(u, \text{orig})\)) and the entry in FU (i.e., the association \(u_{FU}(c\text{sa}, \text{title}, \ldots)\)). The deletion disconnects the file name from the data set.

(4) Upon opening, a unique file union name is created and orig is set to own (this component only has significance in connection with tasks).

(5) The complete set of attributes csa is constructed from the file attributes in attr and from the attributes found in the open statement in the case of explicit opening.

(6) The title is determined by the title option. If the title option does not exist, the title is the file identifier whose declaration contains the file name referred to.

(7) Any I/O action other than opening and closing may change some components of the information associated with u in FU (but not csa and title) and the data set in ES.

Argument Passing

The file name f and the attributes of the argument are passed to the corresponding parameter (Fig. 5.15). There is no other type of argument passing. It follows that the parameter shares with the argument all information linked to the file name, in particular the entries in FU and ES. Files are the only case in PL/I where the attributes are also passed to the parameter and where the attributes declared with the parameter serve only for checking purposes.
where \( n_a \) is the unique name of the argument and 
\( n_p \) is the unique name of the parameter.

**Fig. 5.15 Argument passing**

### Sharing Patterns

(1) **Files which have the same file name** \( f \) **and therefore share all information linked to** \( f\):  
   - (a) **external files having the same identifier**  
   - (b) **files created by the same declaration**  
   - (c) **file parameters and their corresponding arguments.**

(2) **Files sharing the dataset:**  
   - (The same dataset may have more than one title.)
   - All files which have been opened with a title of one and the same dataset share the dataset.  
   - (In all situations quoted by (1) the files of course share the dataset.)
5.6 Procedures

The general diagram for a procedure is:

\[
\text{(id, body, env, bpp, param-list, ret-type)}
\]

\[
\text{DN} \quad \text{n} \quad \text{AT} \quad \text{(attr, env')}
\]

Fig. 5.16 1)

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 identifier id is different from the identifier by which the procedure is referenced in the case where the referencing identifier is a parameter.

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 body of the external procedure to be retrieved. The abstract structure of the body has been given in Fig. 2.3 of chapter 2.2. The environment env determines the interpretation of the body.

The block prefix part is relevant for condition enabling and is inherited with rules similar to those of the environment.

The parameter description list, param-list, and the return type are constructed from the corresponding entry declaration evaluated upon interpretation of the procedure declaration.

The environments env and env' are not the same if the referencing identifier is a parameter.

1) Key to new abbreviations:

bpp block prefix part
param-list parameter description list
ret-type return type
**Argument passing**

The components id, body, env and bpp of the argument are passed to the parameter (Fig. 5.17). The parameter description list and the return type for the parameter are evaluated from the attributes of the parameter when the declaration of the parameter is interpreted.

\[
\text{param-list} \times \text{ret-type} \rightarrow (\text{id, body, env, bpp, param-list', ret-type'})
\]

where: \( n_a \) is the unique name of the argument and
\( n_p \) is the unique name of the parameter

**Fig. 5.17**

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:

\[
\text{param-list} \times \text{ret-type} \rightarrow (\text{id, body, env, bpp, param-list, ret-type})
\]

**Fig. 5.18**
The denotation is \( \mathcal{Q} \).

The attributes are a set of pairs \((\text{id}, \text{param-list})\) where each pair consists of an identifier and a parameter description list. The identifier 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 set. The procedure referenced by the identifier 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, which are not parameters is:

\[
\begin{align*}
\text{id} & \\
\rightarrow & \\
\text{DN} & \\
\rightarrow & \\
\text{n} & \\
\rightarrow & \\
\text{AT} & \\
\rightarrow & \\
\text{(BUILTIN, env)} & \\
\end{align*}
\]

Fig. 5.19

The identifier \( \text{id} \) determines uniquely the builtin function to be evaluated. The actions which have to be performed upon reference to a builtin function, are completely built into the interpretation. The environment \( \text{env} \) is redundant and is only attached to achieve similarity of entries in the attribute directory.

Argument passing

If a builtin function is passed as argument to a parameter then the following entity is created for the parameter (Fig. 5.20):

1) If an argument is a generic reference then the result of the generic selection is passed, i.e. a procedure.
The identifier id is the identifier of the argument. The parameter description list, param-list, is constructed from the entry declaration of the parameter and serves for checking purposes. The attributes attr are the attributes of the parameter and cannot contain expressions. Therefore the environment is redundant.

5.9 Labels

5.9.1 Labels which serve as designations of goto statements

The general diagram is:

```
DN
```

```
(n, id)
```

Fig. 5.21

The denotation is a pair which consists of the unique name and the identifier of the label. The unique name n determines the block activation and the identifier determines the statement within this block activation to which control is passed in case of a goto statement refering to that label. This pair (n, id) is also exactly the information which is represented by the value of a label variable.

No argument passing is to be considered. A dummy label variable is always created to be passed to the parameter, when a label occurs as argument.

The environment attached to the attribute is redundant and serves only to make the entries in the attribute directory alike.
5.9.2 Format labels

The general diagram is:

\[(\text{format-list,env,}\text{st-prefix-p})\]

\[\text{DN} \quad n \quad \text{AT} \]

\[\left(\text{format-list,cond-part},\text{env}\right)\]

\[\text{Fig. 5.2.2}\]

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 denotation is again the information which is represented by the value of label variables. For the same reason as in the previous section no argument passing is to be considered.

5.10 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 they express whether updating a part of an entity means automatically updating of a part of another entity. Consider the example given in Fig. 5.2.3a where both n₁ and n₂ have the same 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. 5.2.3b 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 n₁ and n₂ share y.

![Fig. 5.2.3a n₁ and n₂ share y](image1)

![Fig. 5.2.3b n₁ and n₂ share y](image2)

Chapter 5 has so far not considered tasking. In the sequel the relation 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, of which each task works on its own copy. There are two task local state components, \( \text{AG} \) and \( \text{FD} \), with respect to the state components mentioned in the general diagrams. All other state components mentioned in the diagram are task global. When a task is attached a modified copy of the \( \text{AG} \) and \( \text{FD} \) of the attaching task is made for it.

The modification of the copy of \( \text{AG} \) consists in deleting all generations from the generation lists except the current ones. Let \( \text{AG}_1 \) be an aggregate directory and let \( \text{AG}_2 \) be the modified copy made from \( \text{AG}_1 \) for a task to be attached.
The diagrams in Fig. 5.24 a,b show the versions of a proper variable for the two tasks.

The two versions of the variable obviously share storage via \( \text{gen}_k \). 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 \( \text{gen}_1 \) 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 \( \text{FD} \) consists in changing all occurrences of \( \text{OWN} \) (own) to \( \text{INH} \) (inherited). Since any opening creates an entry \((u, \text{OWN})\), the interpreter can always test whether a file was opened by the current or some mother task. Let \( \text{FD}_1 \) be a file directory and let \( \text{FD}_2 \) be the modified copy made from \( \text{FD}_1 \) for a task to be attached. The diagrams 5.25 a, b show the versions of a file for the two tasks.
The two versions obviously share the information in FU and the dataset. 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.
The two versions are equally valid for the information in 10 and 11. However, the worker can only live with one version. The two versions will no longer share information, and the life with one version will thus be incomplete.

Since the worker will choose a new life, there may be a conflict.
6. PARALLEL TASKS AND I/O-EVENTS

PL/I provides means for specifying that parts of a program can be executed in parallel. These program parts can be tasks attached through a call statement possessing one or more of the options TASK, EVENT, and PRIORITY, or I/O-events attached by read, write, rewrite, delete, or display statements possessing an EVENT option. During the execution of a program the priority information, which is defined by the PRIORITY and TASK option for a task in the attaching call statement, controls in an implementation defined manner which tasks from a set of attached and not yet completed tasks have to be executed first. For I/O-events no relative priority can be specified in the program. In an actual implementation I/O-events will be executed in an implementation defined order according to the availability of devices.

The state of the abstract PL/I machine, as shown in chapter 3 and Fig. 6.1 of this chapter, possesses a major state component PA, the parallel task and event part.

This part PA contains the local state components of all those I/O-events and tasks, which have been attached and are still active, i.e. have not yet been completed. In the following only tasks will be discussed, since I/O-events can be considered as a special, simpler form of tasks. In the interpretation of instructions of an arbitrary task several global state components are relevant. They are shown in Fig. 6.1 as the global state components in box \([2]\). These global state components are the parts \(S\) to \(M\) as listed in \(6/\), p. 2-2.

One of the major problems in tasking is the assignment or access to the same piece of storage from two tasks executed in parallel. The PL/I machine sequentializes the process of interpreting a program in the following basic cycle. A task is selected in an unspecified way and one instruction from this task is executed,
then again a task is selected and one instruction of it executed and so on. Then all possible mergings of the instructions of two parallel tasks are considered. If any implementation would have to treat the instructions of the PL/I machine as atomic and uninterruptable actions of the interpretation of a PL/I program then any assignment or access to the same piece of storage would be defined in the sense of a well-defined and finite set of results. However, the formal definition was in no way intended to bind an implementation in that respect. But on the other hand leaving the subject completely undefined would mean that two tasks could in no way communicate in a defined way and that would frustrate the entire concept of tasking. A possible resolution is to declare certain instructions (e.g. setting of event variables) to be elementary by definition and to declare the other instructions under certain restrictions to be decomposable into more elementary ones.

The selection of the current task is performed by an evaluation of the priorities of the attached and active tasks.

For the interpretation of an instruction various non-global state components are required. These task-local state components are established as required for the specific task to which an instruction belongs. These parts are the parts $TE$ to $C$ as listed in /6/, p. 2-2. Each attached task possesses its local state components in $PA$ (Fig. 6.2). These components for the current task are transferred to box $1$ in Fig. 6.1. For any task the task-local state components are attached to $PA$ through a unique name $n$ or through the selector $s$-main for the major task.

![Fig. 6.2 Sub-parts of the state of a task attached in $PA$]

---

1) E.g. to synchronize two tasks means that they have to communicate at least through certain events.
The computation of the PL/I machine, i.e. the sequence of states starting with an initial state and terminating with an end state is defined in 2.1.6 of /6/. The formula 2-10(15) in /6/ defines the language function \( \Lambda \) which yields for a given state of the PL/I machine the set of successor states.

The flow diagram of Fig. 6.3 shows the various intermediate steps which lead from a given state through the interpretation of an instruction to a successor state.

Fig. 6.3 Interpretation flow in the PL/I machine
The interpretation of an instruction starts with a test in PA whether the
PL/I machine still contains an active task. This test is performed for the initial
state and for each successor state. If PA is empty, i.e. all tasks have terminated,
the computation has successfully terminated.

The flow diagram Fig. 6.3 may be used to produce the sequence of states which
is the computation of the PL/I machine. The sequence starts with \( q_0 \). Any time one
passes through the program point labeled \( i \) during interpretation of the flow dia-
gram a copy of the state is attached to the sequence as the rightmost element.

Because in various cases the language function \( \Lambda \) for a given state yields a
set of successor states, the language function \( \Lambda \) applied to an initial state and
its successor states will yield a set of computations.

If the PA part of the PL/I machine does possess non-completed tasks, the
priority of all attached tasks is evaluated and the task \( t_x \) with the highest pri-
ority is selected. This priority evaluation is implementation-defined. As it also
has to consider I/O-events attached in PA it has to assume an implicit priority for
I/O-events.

When the decision has been made, from which task \( t_x \) the next instruction will
be executed, the PL/I machine has to be prepared for the interpretation of an in-
struction of the task \( t_y \). For this purpose the corresponding subparts of the task
\( t_x \) in PA are transferred to the task-local components of the PL/I machine (box \( 3 \)
in figure 6.1). The major part \( C \) now contains the control part of the task \( t_x \) from
which the next instruction will be interpreted.

The next step in the flow of interpretation is the selection of the next in-
struction in \( C \) for interpretation.

In various places in the interpretation of a PL/I program the language does
not specify in which order certain parts of the program are evaluated. This is the
case e.g. in the evaluation of operands of an expression or in the evaluation of
options of a statement. Thus in the control part of a given task there may exist
several instructions which are candidate for immediate execution. One of these in-
structions, located in the instruction location \( T \) is selected for execution. The
next step in the interpretation flow is the actual interpretation of the instruc-
tion in the instruction location \( T \) of the task \( t_x \). The various possible results of
this interpretation are discussed in /1/.
After the interpretation of the instruction, provided this interpretation was possible and did not result in an error, the new states of the parts of the tasks $t_x$ have to be saved by transferring them back to the task $t_x$ in PA, because the priority evaluation might select a different task for the interpretation of the next instruction.

After the transfer to the $t_x$-part in PA the loop of the interpretation flow continues with the test for the termination of the computation.

It is to be noted that the successor state of a given state is defined as the state of the machine after executing the instruction and after transferring the contents of the task local directories to the respective task in PA.

If an instruction has to delete the task to which it belongs it sets to $\emptyset$ all task-local parts so that after the transfer of the task-local parts to the task in PA this task in PA becomes $\emptyset$ and is thus deleted. If an instruction has to delete a task $t_x$ different from the task to which it belongs the task $t_x$ in PA is set $\emptyset$.

If a new task is to be attached, new task parts are generated and attached with a new task name ten in PA.

For all problems of PL/I which are not concerned with tasks or I/O-events a simplified abstract machine and a simplified interpretation flow can be assumed. The single task machine will not require the parallel action part PA because the task local directories can contain the contents throughout the computation and need not be saved. The interpretation flow simplifies as shown in Fig. 6.4. The computation terminates successfully if $\emptyset$ becomes empty. The control mechanism and different types of instructions that are executed by this mechanism are described in /1/.
7. FLOW OF CONTROL

The following abbreviations will be used throughout this chapter:

- **AG**: aggregate directory
- **arg**: argument
- **AT**: attribute directory
- **bpp**: block prefix part
- **C,c**: control
- **CI,ci**: control information
- **cd**: control dump
- **CS,cs**: condition status
- **CTL**: controlled
- **D,d**: dump
- **den**: denotation
- **descr**: parameter descriptor
- **DN**: denotation directory
- **dyn**: dynamic
- **E,env,e**: environment
- **EI,ei**: epilogue information
- **expr**: expression
- **fct**: function
- **gen**: generation
- **i**: integer value
- **id**: identifier
- **int**: interpret
- **opt**: option
- **pa**: parallel actions
- **param**: parameter
- **pref**: prefix
- **ref**: reference
- **ret**: return
- **s**: storage
- **sc**: statement counter
- **sel**: selector function
- **st**: statement
- **text**: text
- **ten**: unique task or event name
- **tr**: truth value

7.1 Program Initialization

7.1.1 The initial state of the PL/I machine

The interpretation of a program starts with an initial state which essentially is a cleared machine (cf. chapter 3 of /6/). It contains one active task, the main task. The only components which are not cleared and which essentially determine the computation are the following:

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

(b) 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.

(c) The control of the main task. It contains only the instruction `int-program (t,c)` which is executed as first instruction and initiates the complete program interpretation. The two arguments of this instruction are the program t to be interpreted as described in chapter 2 and a call statement c, 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 is implementation dependent, e.g., by control cards or (in the F implementation) by a procedure option MAIN included in the concrete program itself.

The initial instruction handles the program t similarly to the interpretation of a begin block (cf. 7.2), but instead of the statement list of a block the initial call statement is interpreted. Moreover, before the bodies for the declared external entry identifiers are entered as parts of denotations into the denotation directory DN, their text is modified by the so-called prepass.

7.1.2 The prepass

The prepass inspects and modifies all internal and external declarations of controlled and static variables, of file identifiers and all external entry declarations. The purpose is to fix the static properties of these declarations, achieved as follows:

Whenever a block is activated, the identifiers declared in its declaration part are entered with new unique names into the environment $E$ and thereby receive new meaning. Even if the same block is activated twice, the same declaration generates two different unique names and hence completely different meanings for the two block activations. There is no linkage between the information related to these two meanings; indeed there is even no information available as to whether or not the block has been activated before.

Now, there are cases in the language, where this general schema is not sufficient. In these cases the different activations of a declaration or even of different declarations have to share parts of the information constituting their meaning. These cases are the internal and external declarations of controlled and static variables, declarations of files and the external entry declarations. The information to be shared is the denotation and all infor-
This sharing is achieved, before the proper program interpretation, by the prepass adding to each of the relevant declarations a unique name as additional component. Whenever the declaration is activated the unique name is used as denotation for the declared identifier (aggregate name b for a variable, file name f for a file identifier). So, each activation of such a declaration uses the same denotation and thereby the same information accessed by it.

All external declarations of the same identifier get the same unique name from the prepass as denotation. So all external declarations of the same identifier share the same information.

For external entry declarations, the unique names added to the declarations by the prepass are not the denotations themselves; but under these unique names the denotations of the external procedures are entered into the denotation directory immediately after the prepass and before the interpretation of the initial call statement. With these unique names the denotations of the external procedures are available during the program interpretation.

In addition to the modifications of the declarations in the program text, the prepass performs the test for compatibility of external declarations of the same identifier, the allocation and initialization of static variables and the entering of NULL generations into the aggregate directory for controlled variables.

7.2 Block activation

The block structure of a PL/I program leads on interpretation to a dynamic system of nested block activations. Whenever the interpretation of a block (begin block, procedure body or on-unit) starts, a new block activation is established in the state of the PL/I machine; when the interpretation of this block terminates, the previous block activation is re-established. It is a property of PL/I that no block activation is terminated until all nested (i.e., later established) block activations are terminated. The first block activation of a task is established on creating the task and terminated by terminating the task. The current block activation is the last established one which is not yet terminated.
7.2.1 The dump D

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

(a) the environment \( E \),
(b) the epilogue information \( EI \),
(c) the condition status \( CS \),
(d) the dump \( D \),
(e) the control information \( CI \),
(f) 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 components of all active (i.e. established and not yet terminated) block activations which are not the current one are kept in the dump \( D \). The dump is an object manipulated as 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 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 of a task is the null object \( D \).

![Fig. 7.1 The dump](image-url)
When a new block activation is established the local state components of the previous block activation are copied as components into the dump. Thereby automatically the former components of the dump then 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, generally, the nested block activation will modify the inherited state components. (Note: The dump is left unchanged throughout a block activation, except in some cases of abnormal block termination, e.g. goto out of a block). 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.

7.2.2 Interpretation of a begin block

As described in section 2.4.1, a begin block is a proper statement consisting of four components: a declaration part, a procedure body list, a condition part and a statement list. Its interpretation consists of the creation of a new block activation, 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:
Fig. 7.2 Interpretation of a begin block
(a) The local state components are copied into the dump as described in section 7.2.1.

(b) The epilogue information EI (cf. 7.2.4) and control information CI (cf. 7.3) are initialized.

(c) The environment E is updated, as described in section 7.2.3.

(d) The attribute directory AT is updated by entering for each declaration contained in the declaration part of the block its attribute and the current (updated) environment under the unique name n for the identifier.

(e) The block prefix part of the condition status CS is updated by merging the previous one with the condition part of the block.

(f) The denotation directory DN is updated by entering the denotation of each declaration contained in the declaration part of the block under its unique name n. For controlled and static variables and for file identifiers the unique name inserted by the prepass into the declaration is the denotation; for automatic variables it is a newly created unique name. For internal entry declarations, the denotation as described in section 5.6 is constructed from the body to be found in the body list of the block (by the correspondence described in section 2.3.2), the entry identifier itself, the current environment and other components. For external entry declarations, the denotation is found in the denotation directory itself under the unique name inserted by the prepass. (Before interpretation of the initial call statement, the denotations of all external procedures have been entered into the denotation directory). For the other types of declarations a denotation as described in section 5 is constructed and entered into the denotation directory. Additionally for each automatic variable declared in the declaration part of the block, storage is allocated and initialized, and its aggregate name entered into the free-set in the epilogue information (cf. 7.2.4).

(g) The statement list of the block is interpreted. This main part of the block interpretation is described in section 7.3.

(h) 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 EI. Interpretation is continued after termination of these tasks.

(i) The storage of all automatic variables allocated in this block acti-
vatiion (see (f) above) is freed. The information as to which storage is to be freed is found in the epilogue information EI.

(j) The local state components of the previous block activation are copied back from the dump as described in section 7.2.1.

7.2.3 Environment updating

When establishing a new block activation, the environment $E$ is inherited from the previous block activation and updated from the declarations of the new block. This means, that each of these declarations gets a newly created unique name $n$ which is entered into the environment for all qualified names belonging to the declaration (by means of selectors produced by the function sel from the qualified names). These qualified names are for non-variable declarations the one-element-lists of the identifier only, while for variable declarations all identifier lists are computed from the main identifier and the minor structure identifiers which refer to (terminal or non-terminal) components of the variable (see the identifier list component of references in section 2.5.1).

This updating mechanism has the following effects:

(a) All qualified names of the same declaration get the same unique name and thereby refer to the same information in the different directories (cf. chapter 5).

(b) All those qualified names declared in outer block activations which are not redeclared retain their unique names and therefore the same meaning, while for the redeclared ones the unique name in the environment is overwritten by a new one.

(c) Since the environment is a local state component which is handled by the dump mechanism described in 7.2.1, all unique names are available only during the block activation which entered them into the environment. After termination of a block activation the previously overwritten unique names are again available, being popped up from the dump in the environment $E$. 

7.2.3
**Example:** The environments of the following two nested blocks:

A: BEGIN; DCL U, V, 1 X, 2 X1, 2 X2;

```
BEGIN A;
  DCL U, V, 1 X, 2 X1, 2 X2;
END A;
```

B: BEGIN; DCL U, W, 1 X, 2 X1, 2 X3;

```
BEGIN B;
  DCL U, W, 1 X, 2 X1, 2 X3;
END B;
```

may be represented by the following tables:

<table>
<thead>
<tr>
<th>qualified name</th>
<th>unique name</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>n_1</td>
</tr>
<tr>
<td>V</td>
<td>n_2</td>
</tr>
<tr>
<td>X</td>
<td>n_3</td>
</tr>
<tr>
<td>X.X1</td>
<td>n_3</td>
</tr>
<tr>
<td>X.X2</td>
<td>n_3</td>
</tr>
</tbody>
</table>

Environment of A (current as long as A, but not B, is active):

Environment of B (current as long as B is active):

Note, that in this example during the activation of B, the component
X.X₂ of the structure X declared in block A is available by n₃, while X and X.X₁ lead by n₆ to the structure X declared in block B.

The environment of A has the structure given in the following figure:

```
    sel(<U>)    sel(<V>)    sel(<X>)    sel(<X,X₁>)    sel(<X,X₂>)
     /   \      /   \      /   \      /   \      /   \     
   n₁   n₂  n₃   n₃   n₃
```

**Fig. 7.3**

### 7.2.4 The epilogue information

The epilogue information EI is a local state component which for each block activation contains all information necessary for its correct termination. It consists of the following components:

(a) 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 be 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.

(b) 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.

(c) The block-activation-type. A return statement has to perform different actions depending on whether the current block activation is a begin block, a normal procedure, a task or an on-unit. To recognize this distinction the epilogue information contains a component which is one of the elementary
objects BLOCK, PROC, TASK, ON. This component is set when establishing a block activation and never changed.

(d) The function generation. If a procedure is activated by a function reference, then before the call of the procedure a dummy variable is created for the function value to be returned to (and storage allocated for this dummy variable). The allocation is performed using the return type of the entry declaration. The generation of this dummy variable is reserved in the epilogue information of the called procedure. A return statement in the procedure assigns the return value to this dummy generation. 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 \( PT \) is the null object \( \emptyset \) in procedures activated by call statements instead of function references (and in nested begin blocks).

(e) The main procedure flag. If a return statement has to terminate the main procedure, i.e., the procedure activated by the initial call statement 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 \( T \) in the initial state and reset to \( \emptyset \) by all later procedure calls. Like the function denotation this flag is inherited into the nested block activations of begin blocks.
7.3 Sequential Interpretation of a Statement List

This section describes the interpretation of a statement list, e.g. the statement list of a block. A statement list is generally interpreted by sequentially interpreting the individual statements of the list in their given order, thereby omitting any entry points which occur (cf. 2.2).

7.3.1 General mechanism

The sequential interpretation is governed by the control information CI, which is one of the six local state components. One of the components of CI, the text component, contains the statement list to be interpreted. A second component of CI, the statement counter, is an integer value specifying which element of the statement list is currently being interpreted.

The interpretation of a statement list is started by the instruction int-st-list(t) (cf. 5-16(23) of /6/), which copies the statement list t to be interpreted into the text component of CI, initializes the statement counter to 0, and starts the interpretation of the first statement with the instruction int-next-st. Whenever a new statement is to be interpreted (e.g. after completion of the interpretation of the previous statement) the instruction int-next-st (cf. 5-16(24) of /6/) increases the statement counter by 1 (unless it points to the last statement in the text component), and starts the interpretation of the statement to which the increased statement counter points. This is done by the instruction int-st(st_{i+1}) where st_{i+1} is the (i+1)-st element of the text component.

The described process may be illustrated by the following flow chart:

Fig. 7.4 Text and statement counter
The described mechanism is not quite sufficient, since the interpretation of a single statement (the sc-th element of tx, cf. Fig. 7.5) may itself include the interpretation of a statement list, namely if its proper statement is a begin block, group, statement list or if-statement (cf. 2.4). In this case the text component of CI and the statement counter would be overwritten during the interpretation of the single statement, and so this information, necessary after completion of the statement, would be destroyed. In the case of a begin block there is no problem, since the new block activation established by its interpretation reserves the old control information CI in the dump D anyway (cf. 7.2.1) and re-installs it after its completion.

For all those cases, where during the current block activation a nested statement list is to be interpreted by the interpretation of a single statement, the control information CI contains a third component, the control dump, which works similarly to the dump D (cf. 7.2.1). Whenever a statement list is going to be interpreted, the complete control information CI and the control C are copied as components into the control dump, before the actions described above are performed, and
they are recopied into the state of the machine from the control dump after completion of all actions described above. So Fig. 7.5 has to be completed as follows:

As with the dump D (cf. 7.2.1), the former control dump becomes a part of the new control dump by copying the control information CI into it, and thereby all contained information is pushed down. Conversely, after recopying CI from the control dump, all information is popped up again.

The complete control information CI is initialized to Ω on creation of a block activation, since for the first actions of a block activation (the prologue, actions a to f of Fig. 7.2) none of the information in CI is needed.
The interpretation of the statement list of the block (action g in Fig. 7.2) installs CI as described above and finally recopies $\emptyset$ into CI. So, for the last actions of a block activation, the epilogue (i.e., actions h to j in Fig. 7.2) again CI is the null object $\emptyset$. That means that $CI = \emptyset$ is characteristic for the prologue and epilogue of a block activation and $CI \neq \emptyset$ for the interpretation of the statement list of a block activation.

7.3.2 Structure of the control information CI

The structure of the control information follows from the complete description of the mechanism and is shown in Fig. 7.7.

Fig. 7.7 The control information CI

In this figure st-list and sc are the current statement list and statement counter, while all subscripted elements are the stacked ones. st-list$_1$ is the complete statement list of the current block activation, whose sc$_1$-th statement is being interpreted. This sc$_1$-th element of st-list$_1$ contains as its proper statement
the statement list \( st-list_2 \), whose \( sc_2 \)-th element is being executed; and so on. Thus, the control information denotes, within the current block activation, exactly the innermost statement currently being interpreted within the structure of nested statement lists.

Note that, given the outermost statement list \( st-list_1 \), the sequence of statement counters \( sc_1, sc_2, \ldots, sc_n, sc \) would be sufficient to localize exactly the innermost statement currently being interpreted in this structure of nested statements. The only exception is that an if-statement may contain two statements, the then- and the else-alternative, and it is not clear whether the if-statement itself or one or other of its alternatives is denoted (or if both alternatives are statement lists, in which of them the innermost statement is contained); but this difficulty might be solved by adding e.g. the truth value \( T \) or \( F \) as statement counter to the sequence \( sc_1, \ldots, sc_n, sc \), when starting the interpretation of the then- or else-alternative, respectively. This possibility, of denoting a statement exactly in the structure of nested statement lists by a sequence of statement counters including truth values, will be used by the interpretation of the goto statement (cf. 7.4.2).

Therefore with the exception of if-statements, the text components \( st-list_2, \ldots, st-list_n, st-list \) in \( CI \) are redundant. Nevertheless it is very convenient to have immediately available the current statement list \( st-list \), to which the current statement counter \( sc \) refers, as described in the beginning of section 7.3.1.

Since the control \( C \) is also stacked into the control dump when the interpretation of a new statement list is started, the current control component \( C \) of the machine state contains only the instructions concerned with the interpretation of the innermost statement list. In fact, it contains only the instructions concerned with the current statement and, at the top, the instruction \( \text{int-next-st} \) described in section 7.3.1. The instructions concerned with outer statements, e.g. the iteration control of a group while the statement list of the group is interpreted, are contained in the stacked control components \( c_n, \ldots, c_1 \). The control component \( c_0 \) contains in particular the instructions concerned with the epilogue of the current block activation.
7.4 Goto Statement

A goto statement consists essentially of a reference (cf. fig. 7.8), which refers (if it is correct) either to a statement label constant or to a scalar label variable, whose value is the denotation of a statement label constant.

The interpretation of a goto statement transfers control to the statement labeled by the statement label constant. That means it has to change all those state components of the PL/I machine which are concerned with the flow of control to the status they would have had if the labeled statement had been reached by the normal sequential flow of control as described in section 7.2 and 7.3. These state components are: the control C, the control information CI and the dump D (and hence also the other local state components).

7.4.1 The denotation of a label constant

The situation to which the state of the PL/I machine is changed by a goto statement is uniquely determined by:

(a) a block activation (cf. 7.2.1),
(b) a statement in the nested structure of statement lists within the statement list of that block activation (cf. 7.3.2).

Note that determining a statement within the complete program text is not sufficient, since the block to which this statement belongs may be activated more than once.

The above two characteristics are uniquely specified by the following information:

(b) The statement within a block activation by the identifier id of the statement label constant, since within one block (except nested blocks) no two statements may be labeled by the same identifier.
(a) The block activation by the unique name $n$ associated with id by its environment $E$, since an identifier id occurring as a label constant is declared local to the containing block (cf. 2.3) and gets a new unique name $n$ for each activation of that block. (Note: If $n$ is inherited into the environment of a nested block activation, since id is not redeclared (cf. 7.2.3) id cannot occur as statement label in the statement list of the nested block activation).

Therefore in the formal definition, the denotation of a label constant is an object which consists of an identifier id and its unique name $n$ in the environment of a certain block activation:

![Fig. 7.9 Denotation of a statement label constant](image)

One should note the following two remarks:

1. If the reference in a goto statement refers to a label constant, this is in the scope of the declaration of its identifier id. Thus the corresponding unique name $n$ might have been taken from the current environment and the denoted block activation is the current one or the last one in which id occurs as statement label. So, in this case, the identifier id alone would be sufficient, and the unique name is redundant. The identifier alone is not sufficient if the reference in a goto statement refers to a label variable, where a label constant was assigned to the label variable in a former block activation and the identifier id of this label constant was redeclared afterwards. In order to transfer control to the assigned label constant correctly, the identifier id has in this case to be qualified by its unique name $n$. In order to handle both cases uniformly, the pair $(id, n)$ is used as denotation of a label constant and as value of a label variable and is in either case returned by evaluation of the reference in the goto statement.
Example:

```plaintext
A:BEGIN;
  DECL V LABEL;
  V = L;
  L:statement-1;
  ...
B:BEGIN;
  ...
  L:statement-2;
  GOTO V;
  ...
  END B;
  ...
END A;
```

In this example the identifier L has a unique name \( n_1 \) in an activation of block A and another unique name \( n_2 \) in the nested activation of block B. The assignment statement \( V = L \) in block A assigns the pair \( (L,n_1) \) to \( V \). The goto statement \( GOTO V \) in block B transfers control to the label denoted by \( (L,n_1) \), i.e. to L in block A (the first label L), though in block B there is another label L (the second label L, denoted by \( (L,n_2) \)). A similar but not so obvious situation occurs if the same block A is activated twice by two nested block activations, L is assigned to V in the first block activation and the goto statement is interpreted in the second one.

(2) In principle the identifier id in the denotation of a label constant is redundant, since no two different (main) identifiers have the same unique name \( n \) (cf. 7.2.3). But the identifier id is necessary for scanning statement lists for a statement labeled by id, and since the formal definition has no facility which (inversely to the environment) allows the identifier id corresponding to a unique name \( n \) to be found, the identifier id is copied into the denotation of a label constant.

### 7.4.2 Scanning a statement list for a label

The first action of a goto statement (after evaluation of the label reference) is to scan the current statement list, i.e. the text component of the control information, for a statement labelled by id (the identifier of the label denotation). Thus all statements which are elements of the list and all statements contained in if-statements and nested statement lists (non-iterated DO-groups in the concrete program, cf. 2.4.2), in so far as they are not contained in nested blocks or (iterated) groups, have to be inspected.
This scan is performed by an algorithm which, if it is successful, returns a sequence of statement counters (including truth values denoting the alternatives of if statements) localizing the labeled statement within the statement list as described in section 7.3.2. If it is unsuccessful it returns the null object \( \varnothing \). The algorithm is described by the flow charts in Fig. 7.10.a (scanning a statement list; cf. the function search-1, 5-29(46) of /6/) and Fig. 7.10.b (scanning a single statement; cf. the function search, 5-28(48) of /6/). Since a single element of the statement list may again contain a statement list to be scanned, the algorithm is given by recursively defined functions.

---

**Fig. 7.10a** Scanning a statement list for a label

---

7.4.2
Fig. 7.10b  Scanning a statement for a label
7.4.3 Goto within the current statement list

The simplest case of a goto statement is a goto to one of the statements of the current statement list. This is the case if:

(a) the scan described in section 7.4.2 applied to the current statement list is successful and returns a statement counter list consisting of a single integer value \( i \);

(b) in the current environment \( n \) is the unique name associated with \( id \).

In this case only the statement counter is changed to \( i \), the interpretation of the \( i \)-th element of the current statement list is started and then it is continued in the regular way as described in section 7.3.1. No other state component is changed (cf. the first case of the instruction goto-jump, 5-29(50) of /6/).

A goto statement leads into a nested statement within the current statement list if:

(a) the scan described in section 7.4.2 is successful (and returns a statement counter list consisting of more than one element);

(b) in the current environment \( n \) is the unique name associated with \( id \).

In this case, starting with the current \( CI \), one has to build up the control information, which would have appeared in the state of the machine if the labeled statement had been reached by the normal flow of control (cf. 7.3). This is performed, step by step, by the instruction goto-jump (cf. 5-29(50) cf /6/).

If the statement counter list \( sc-list \) returned by the scanning algorithm consists of integer values only, i.e. no if-statement alternatives are involved, one has to continue to build up the control information \( CI \), as described in Fig. 7.7, to the top by going step by step into the nested statement lists deter-
mined by sc-list. A single step first changes the statement counter to the first integer value of sc-list, say i; then it stacks complete CI and a control consisting of the single instruction int-next-st (since, after completion of the nested statement list, the (i+1)st statement of the current statement list is immediately to be executed) into the control dump; then it copies the proper statement of the i-th element of the current statement list (which in our case is again a statement list) into the text component of CI; finally it removes the first integer value from sc-list. The current statement counter sc in the control information is left unchanged; it is irrelevant, since the first action of the next step will adjust it to a value specified by sc-list. This step is repeated until sc-list is finally reduced to a single integer value; then the simplest case described in the beginning of this section applies. The described step is performed by the second case of the instruction goto-jump (cf. 5-29(50) of /6/). This algorithm is demonstrated by Fig. 7.12.

Fig. 7.12 Goto into nested statement lists (not into if-statements)
If however, the statement counter list sc-list returned by the scanning algorithm also contains truth values, the goto statement leads also into if-statement alternatives. In this case the same steps as described before are performed as long as the first element of sc-list is an integer value. Now, the last such step (i.e., a step working with sc-list starting with an integer value, say i, followed by a truth-value, say tr) copies as i-th element of the current statement list an if-statement instead of a statement list into the text component of CI. For the next step there are two possibilities:

1. If tr is not the last element of sc-list, then tr is removed from sc-list, and the if-statement in the text component of CI is replaced by the proper statement of its then- or else-alternative depending on the value of tr, T or F, respectively. This is a statement list (if the next element of sc-list is an integer value) or an if-statement (if the next element of sc-list is a truth value). In the former case, the text component of CI is correct again and one can continue as described above. In the latter case the replacement of the if-statement by one of its components is repeated, until one finally comes back to an integer value in sc-list (i.e., a statement list in the text component of CI) or to the last element of sc-list, which is handled as described below.

2. If tr is the last element of sc-list, the previous step has stacked one level too many into the control dump (and this fact cannot be justified by following steps which replace the if-statement in the text component of CI by a statement list in one of its alternatives). Instead of stacking CI into the control dump and copying the if statement into the text component, the if-statement alternative determined by tr should be interpreted. Therefore the last level of stacking in the control dump is cancelled and the flow of control is started with the right alternative of the if-statement.
7.4.4 Goto out of the current statement list

If either the scan described in section 7.4.2 is unsuccessful within the current statement list or n is not the unique name associated with id in the current environment, then the goto statement leads out of the current statement list.

Since a goto statement leading from outside into a block or (iterated) group is not allowed, it has to lead into one of the statement lists and block activations represented in the control dump and dump respectively. In doing so, each of the groups and block activations left by the goto statement has to be terminated.
This is performed by terminating the statement lists in the control dump and the block activations in the dump level by level until one comes to a statement list, in which the scan of 7.4.2 is successful and an environment by which n is associated with id. Then the goto mechanism described in section 7.4.3 is performed. This is demonstrated by Fig. 7.14 which describes the instruction goto-search (cf. 5-28(47) of /6/).

Fig. 7.14 Interpretation of a goto statement

7.5 Procedure Call

A procedure call establishes a block activation of a procedure body by a call statement or function reference. The current section will describe all those actions performed during a procedure call which differ from the actions performed during a begin block activation as described in 7.2.

Both call statement and function reference specify an entry identifier and an argument list, which is a list of expressions (a function reference specifies the entry identifier in the form of a one-element list since it is syntactically
a special case of a reference to a variable, cf. 2.5.1; a call statement may additionally have a parallel action option specifying a task call, but this is not considered in this section).

**Fig. 7.15 Call statement and function reference**

The specified entry identifier gives, via the environment $E$, attribute directory $AT$ and denotation directory $DN$ (Fig. 7.16, cf. chapter 5), access to an entry attribute (Fig. 7.17) and an entry denotation (Fig. 7.18).
A procedure call differs from a begin block activation in the following main points:

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

(b) Instead of the current environment \( E \) the environment of the activation of the block in which the entry declaration occurred is inherited into the new block activation. This environment is contained in the denotation of the entry identifier. The same is valid for the block prefix part of the condition status \( CS \).

(c) 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 section 7.5.1.

(d) In the case of a function reference a value is returned after termination of the block activation as described in section 7.5.2.

(e) The interpretation of the statement list is not started at its beginning, but an entry point somewhere in the statement list. This entry point is specified by the actual entry identifier contained in the entry denotation. Note that this identifier need not be identical with the identifier given in the call statement or function reference (for if the latter is a parameter, the actual entry identifier is the entry identifier of the argument passed to this parameter). The start of the interpretation of the statement list is performed by the same mechanism as that of a goto statement within the current statement list (cf. 7.4.3 and 7.4.2), the only difference being that the scanning algorithm searches for an entry point with the entry identifier \( id \) instead of a statement with a label \( id \).

7.5.1 Argument passing

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

(a) the \textbf{argument expressions} in the call statement or function reference,

(b) the \textbf{unevaluated parameter descriptors} in the entry attribute (if \textbullet{} is specified instead of a descriptor list, all descriptors are assumed to be \textbullet{}),

(c) The \textbf{evaluated parameter descriptors} in the entry denotation (if \textbullet{} is speci-
fied instead of a descriptor list, all parameter descriptors are assumed to be *).

(d) the **parameter identifiers** in the entry point specified by the actual entry identifier in the statement list of the procedure body,

(e) by means of the parameter identifiers the **parameter declarations** in the declaration part of the procedure body.

For each single parameter the following actions are performed:

(1) Before the call, i.e. in the old block activation, the decision is made as to which of the following three types of passing is to be performed. This decision is based on the argument expression and the unevaluated parameter descriptor, so that it can be done without expression evaluation (i.e. "at compile time").

1) **Passing of denotation** (direct passing). This is performed if the argument expression is a simple reference (i.e. unqualified and unsubscripted) referring to a controlled variable, entry or file declaration and if either this declaration and the parameter descriptor match or the parameter descriptor is an asterisk *.

2) **Passing of generation**. This is performed if the argument expression is any reference to a variable (or to a sub-part of a variable) and if either the data attributes of this variable (or sub-part) and those of the parameter descriptor match or the parameter descriptor is an asterisk * (provided that passing of denotation does not apply).

3) **Passing of value** (dummy variable passing). This is performed in all other cases, if the parameter descriptor is a variable (not entry or file) descriptor or an asterisk *.

(2) Before the call, the argument is evaluated resulting in an object consisting of a denotation and a type designator (an elementary object specifying the type of passing and the type of the parameter: CTL, ENTRY, FILE, GEN, DUMMY).

![Fig. 7.19 Structure of passed argument](image-url)
This argument depends on the argument expression and the evaluated parameter descriptor. The evaluation depends on the type of passing:

1) Passing of denotation. In this case, in general, the denotation of the argument is itself the denotation of the controlled variable, entry or file identifier referenced by the argument expression, which is a simple reference (an exception is valid only for generic or builtin arguments, where the generic selection is to be made during argument evaluation and the denotation of the selected member of the generic family is taken).

2) Passing of generation. In this case the generation referenced by the argument expression, which is any reference, is evaluated. If the data attributes of this generation match the data attributes of the evaluated parameter descriptor, this generation is entered into the aggregate directory AG under a newly created unique aggregate name b, which then becomes the denotation of the argument.

3) Passing of value. In this case first the data attributes of the result of the argument expression are determined. Then a dummy variable is created with the data attributes of the evaluated parameter descriptor, completed where necessary by the data attributes of the argument expression. For the dummy variable a unique aggregate name b is created, storage is allocated and the corresponding generation entered under b into the aggregate directory AG. Finally, the argument expression is evaluated and assigned to the dummy variable. The aggregate name b becomes the denotation of the argument.

(3) After the call, i.e. in the block activation established by the procedure call, the parameter identifier, as all locally declared identifiers, is in all cases associated with a unique name n in the environment and the attributes of the parameter declaration are entered into AT. The argument evaluated before the call as described is passed to the called block activation. Its denotation is, after testing of the argument against the parameter declaration, entered into the denotation directory DN under the unique name of the parameter.

```
Fig. 7.20 Connection of parameter identifier with denotation and attribute
```
So, generally, in the called block activation the parameter identifier is connected via environment, attribute directory and denotation directory with the attributes of the parameter declaration and the denotation resulting from the argument evaluation.

There are two exceptions from this general rule: For file parameters the attributes are also taken over from the argument. For controlled arguments passed to non-controlled parameters after the call, the last generation instead of the complete generation list in AG 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:

```plaintext
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;
```

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 environment, denotation directory and storage to the value 0 are as demonstrated in the following figure:

![Fig. 7.21 The three types of argument passing](image.png)
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 in their attributes in AT as far as it is allowed.

7.5.2 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 an operand is to be returned.

For this purpose the return type specifying the data attributes of this operand is given in the entry declaration. This return type is evaluated (if it contains an expression specifying a string length) and taken over into the entry denotation. A second return type is specified in the entry point within the statement list of the body (cf. 2.2).

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

(1) Before the call, a dummy variable is created and allocated according to the evaluated return type contained in the entry denotation. The generation of this dummy variable is entered as function generation component into the epilogue information EI created for the called block activation (cf. 7.2.4).

(2) In the prologue of the called block activation the two return types are tested for matching (the return type of the function generation in EI against the return type in the entry point).

(3) A return statement in the called block activation (or in a nested one) assigns the value of its expression to the dummy variable, whose generation is found in the epilogue information. Since the return statement may occur in a nested begin block activation, the function generation is inherited into the epilogue information of all nested begin block (but not procedure) activations.

(4) After termination of the called block activation, the operand is extracted from the dummy variable and the dummy variable is freed.
7.5.3 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 by 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, the end of the statement list of a procedure body is handled as a return statement without an expression specified.

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 all necessary information to be available, the epilogue information ET contains the function generation (or 0 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. 7.2.4).

Using this information, a return statement works as described by Fig. 7.2.2.
### 7.5.3 Return Statement

**Fig. 7.22**

- **Error?**
  - No → Continue with block-epilogue of Fig. 7.2.
  - Yes → End.

- **Task?**
  - Yes → Continue with block-epilogue of Fig. 7.2.
  - No → Continue with block-epilogue of Fig. 7.2.

- **Procedure?**
  - Yes → Continue with block-epilogue of Fig. 7.2.
  - No → Continue with block-epilogue of Fig. 7.2.

- **Begin block?**
  - Yes → Raise FINISH condition.
  - No → No operation.

- **Return-expression specified?**
  - Yes → Return result.
  - No → No operation.

- **Called by?**
  - Yes → Continue with block-epilogue of Fig. 7.2.
  - No → Continue with block-epilogue of Fig. 7.2.

- **Continue with?**
  - General expression.
  - Assign value to symbol.
  - Evaluate expression.
8. ALLOCATION, ASSIGNMENT AND EXPRESSION EVALUATION

8.1 Allocate Statement and Free Statement

8.1.1 Allocate Statement

An allocate statement specifies a list of allocations (cf. 2.4.3) which are executed in order from left to right. Fig. 8.1 shows the structure of the specification of a single allocation, with an indication as to which components are significant for which types of allocation.

![Fig. 8.1 Structure of specification of an allocation](image)

The identifier is the identifier of the variable to be allocated. Three types of allocation have to be distinguished.

8.1.1.1 Allocation of controlled variables

The allocation of a controlled variable proceeds in the following steps:

1. The data attributes to be used for allocation are evaluated. These data attributes are those 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 attributes are specified in the allocate statement, then the declared data attributes are evaluated in the environment in which the controlled variable was declared,
   
   b) if allocation attributes are specified in the allocate statement, then extents are taken.
from the attributes in the allocate statement if specified thereby.

from the data attributes of the current generation of the variable if

from the declared data attributes if left unspecified in the statement.

Extent expressions specified in the statement are evaluated in the current
environment, extent expressions taken from the declaration are evaluated in
the environment of the block activation in which the variable was declared.

2. A pointer is selected using the evaluated data attributes, the declared den-
sity, and the allocation state of 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, the evaluated data attributes,
and the density. This generation is put on top of the list of generations
associated with the variable in the aggregate directory $AG$ (cf. 5.3).

4. The variable is initialized, using the generation just defined, the initial
set specified in the allocate statement, and the initial set specified in
the declaration of the variable. The elements of the initial sets are pairs
consisting of an unsubscripted reference to the variable to be initialized,
and an initial specification or a call statement. The references must be
scalar references, or array references to arrays of scalars. The two sets
are merged with the rule that an element of the set specified in the state-
ment overrides an element of the set specified in the declaration if the
reference parts of the two elements are equal.

The evaluation of an initial specification results in a sequence of operands.
These operands are assigned sequentially to the scalar parts of the associat-
ed reference. This process stops if the operand list is exhausted, or if the
scalar parts of the reference are exhausted, i.e., the number of elements
of the operand list may be greater or smaller than the number of scalar
parts of the associated reference.

If the second part of an element of an initial set is a call statement, the
call is performed.

The initialization is completed if all elements of the merged initial sets
have been treated in the above way.
8.1.1.2 Allocation of based variables in main storage

The pointer reference may or may not be present in the statement. The allocation proceeds in the following steps:

1. If a pointer reference is given in the statement, the generation associated with this reference is evaluated. If not, the generation associated with the pointer reference given in the declaration of the based variable is evaluated in the environment of the block activation in which the declaration was made.

2. A pointer is selected using the evaluated data attributes associated with the unique name of the based variable in the denotation directory $DN$ (cf. 5.4), the declared density of the based variable, and the allocation state of 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. An operand is formed from the pointer value and assigned to the generation defined above.

4. The pointer is also added to the free-set associated with the current task. The free-set is a set of pointers identifying storage parts that have been used for allocation via based variables in $S$. It is used to free all the storage at termination of the task.

5. A generation is formed from the pointer, the evaluated data attributes and the density of the variable. This generation is used for initialisation of the based variable. The initialisation proceeds as described for controlled variables. No initial set, however, is given in the allocate statement for a based variable. Only the initial set given in the declaration of the based variable is used.

8.1.1.3 Allocation of based variables in areas

The pointer reference may or may not be present in the statement. The allocation proceeds in the following steps:

1. The evaluation of the pointer (or offset) reference given in the allocate statement, or in the declaration, proceeds as for the allocation in main storage.
2. The generation associated with the area reference is evaluated.

3. An offset (i.e., a pointer relative to an area, cf. 4.2.7) is selected using the evaluated attributes and the density of the based variable, and the allocation state of the area identified by the area generation. A test is made whether the allocation in the area is possible (cf. 4.2.3). If not, the AREA condition is raised.

If the allocation is possible, the offset is added to the allocation state of the area.

4. An operand is formed from the offset value. If an offset reference was specified (see point (1)), this operand is assigned using the generation obtained in step 1.

If the reference is a pointer reference, the operand is converted to a pointer operand (cf. 4.2.7). The converted operand is then assigned.

5. The offset is converted to a pointer (cf. 4.2.7) and used to form a new generation together with data attributes and density of the based variable. The generation is used for initialisation, which proceeds as for allocations made 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.

8.1.2 The free statement

A free statement specifies a list of freeings which are executed in order from left to right. Fig. 8.2 shows the structure of the specification of a single freeing with the indication as to which components are significant for which type of freeing:
8.1.2.1 Freeing of controlled variables

The following actions are taken:

1. The top-most generation of the list of generations associated with the controlled variable in the aggregate directory is deleted.

2. The pointer contained in the pointer part of this generation is deleted from the allocation state of the main storage $S$.

8.1.2.2 Freeing of based variables in main storage

The pointer reference may or may not be present in the statement. The following actions are taken:

1. If a pointer reference is given in the statement, this reference is evaluated. If there is no pointer reference in the statement, the pointer reference given in the declaration of the based variable is evaluated. The result is an
operand which specifies a pointer.

2. This pointer is deleted from the allocation state of the main storage $S$. A test is made as to whether this pointer actually was present in the allocation state, and whether the associated storage part could have been allocated by the based variable specified in the free statement.

3. The pointer is also deleted from the free-set belonging to the current task, thus preventing any attempt to free the associated storage part a second time at task termination.

8.1.2.3 Freeing of based variables in an area

The pointer reference may or may not be present in the statement. The following actions are taken:

1. A pointer value is evaluated as for the freeing of based variables in main storage.

2. The generation associated with the area reference is evaluated.

3. The pointer is converted to an offset using the area generation (cf. 4.2.7).

4. The offset is deleted from the allocation state of the area identified by the above generation. A test is made as to whether this offset actually was present in the allocation state, and whether the associated storage part could have been allocated by the based variable specified in the free statement.
8.2 Assignment Statement, Expression Evaluation, Reference to Variables

3.2.1 Expansion of aggregate assignment statement

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.

Assignment statement:

\[
\begin{align*}
\text{s-lp} & \quad \text{expr} \\
\text{ref}_1 & \quad \text{ref}_2 & \quad \ldots & \quad \text{ref}_n \\
\end{align*}
\]

Fig. 8.3 Structure of assignment statement

The references in the left part are references to variables and/or to pseudo variables (pseudo variables are discussed in 8.2.5).

In order to simplify discussion, the term data attributes of the reference to a variable will mean the data attributes 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 data attributes of the reference are array or structure data attributes, 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 data-attributes \(eda_1\) of the left-most reference in the left-part. (If the left-most reference happens to be the reference to a pseudo variable, the data attributes of the first argument are taken.) It proceeds as follows:

- If \(eda_1\) is scalar, then the assignment statement is interpreted.
- If \(eda_1\) is non-scalar, either all references in the left part must be arrays or all references in the left part must be structures. For each integer \(i\) which determines an immediate sub-part of \(eda_1\) (cf. 4.2.4), proceeding sequentially from the smallest to the greatest integer, the following actions are taken:
1) The text of the assignment statement is modified, as determined by eda1 and the integer i (see below),

2) 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 data attributes eda1 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 eda1 is an array data attribute with the same number of dimensions and the same bounds, then the reference is replaced by the reference to the ith sub-part of the array.

6) If the expression is a structure reference to a variable and eda1 is a structure data attribute with the same number of elements, then the reference is replaced by the reference to the ith sub-part of the structure.

7) If the expression is a structure reference to a variable and eda1 is an array data attribute, then the reference is left unchanged.

8) If the expression is a reference to a builtin 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 builtin function.
(9) All cases not mentioned above are erroneous.

8.2.2 Assignment statement BY NAME

Assignment statements given the BY NAME option are expanded in a different way. The difference to the non-BY NAME expansion arises when \( \text{eda}_1 \) is a structure attribute. Then the information given to the modifying function is not the integer of a sub-part, but the sub-aggregate name \( \text{id} \) identifying the sub-part in the left-most 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 \( \text{eda}_1 \) is a structure attribute, then the reference is replaced by the reference with \( \text{id} \) appended as name qualifier, provided that a sub-part with name \( \text{id} \) 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.

8.2.3 Scalar assignment

A scalar assignment other than an area assignment statement is executed by

(1) evaluating the sub-generations associated with the references in the left-part (cf. 4.2.6 and 8.2.7), in order from left to right.

(2) evaluating the right part expression, which results in an operand (cf. 8.2.6),

(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 attributes (cf. 8.3.2). 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 \( \text{op}' \) and the pointer part of the generation be \( \text{p} \) then a new storage part \( S' \) is created on assignment:

\[
S' = \text{el-ass}(p, s-\text{vr}(\text{op}'), \underline{2})
\]
8.2.4 Area assignment

All references in the left-part and also the right-part expression of an area assignment statement are area references. The following actions are taken on interpretation:

1. The sub-generations associated with the left-part references are evaluated in order from left to right.

2. The area operand associated with the right-part reference is evaluated,

3. The assignments of the right-part area to the areas identified by the left-part generations are executed in order from left to right.

Let the pointer part of a left-part generation be \( p_1 \), allst the allocation state of the right-part area and \( v_{r_1} \) the value representation associated with the \( p_1 \)-part of the right-part area, where \( p_1 \) is a member of allst. allst as well as the \( v_{r_1} \) are retrievable from the area operand. An area assignment is interpreted by:

1. Setting the allocation state of the area \( p_1(S) \) to be empty,

2. Testing whether all pointers \( p_1 \) contained in the allocation state allst are such that \( p_1 \cdot p_1(S) \) is a storage part usable for allocation, i.e., that \( p_1 \) is applicable to \( p_1(S) \) and that \( p_1 \cdot p_1(S) \) is independent of the allocation state part of \( p_1(S) \). If this condition is not satisfied, the AREA condition is raised,

3. For each \( p_1 \) in the allocation state allst, changing the storage part in two steps by
   a) \( S' = el\text{-alloc}(p_1, P_1(S)) \)
   b) \( S'' = el\text{-ass}(p_1 \cdot p_1, v_{r_1}, S') \)

After the assignment, the area \( p_1(S) \) has the same allocation state as the right-part area, and all storage parts used for allocation in the one area, have associated the same value representation as the corresponding storage parts in the other area. 'Corresponding' here means having the same offset with respect to the containing area (cf. 4.2.7).
### 8.2.5 Assignment to pseudo variables

Pseudo variables are means for assigning to various kinds of storage parts which are otherwise inaccessible. It is, for example, possible to assign to a sub-part of a scalar string variable via the pseudo variable SUBSTR.

A reference to a pseudo variable, occurring 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 the 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 attributes depending on the pseudo generation.

**Example:** Let UNSPEC(X(7)) be a reference to the pseudo-variable UNSPEC. The evaluation of the reference involves the evaluation of the sub-generation associated with the reference X(7). Let this sub-generation be gen, then the resulting pseudo generation consists of

1. the abstract identifier corresponding to UNSPEC,
2. the generation gen.

The evaluation of a right part expression results in an operand, say op. On assignment of op via the pseudo generation it is converted to a bit string operand, where the length depends on the data attribute part of gen in an implementation defined way, and the value representation part of the converted operand is assigned to the storage identified by the pointer part of gen.
8.2.6 Expression evaluation

The evaluation of an expression results in an operand (cf. 4.1). In many instances expressions occur where the meaning of the various identifiers used in the expression is determined in one block activation, but which have to be evaluated in a different block-activation where the meaning of the identifiers may have changed. This occurs, e.g., when the base of a defined variable is evaluated on reference to the defined variable. Expressions have got their meaning in the block activation in which the defined variable has been declared. This meaning, however, can be retrieved when the environment part of this block activation is known. Expressions are therefore always evaluated by explicitly stating an environment part which has to be used for retrieving the meaning of the identifiers in the expression. We shall say that an expression \( t \) is evaluated in a certain environment \( E \), which need not be the environment part of the current block activation.

In the present chapter the various possible forms of expressions are enumerated and discussed. An expression may be one of the following (cf. 2.5):

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. 8.3).

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. 8.3).

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 already has the form of an operand, so that it is the direct result of the evaluation.

5. A reference. A reference may refer to a variable, to an entry or generic name, to a builtin function, or it may be a label or a format label. References to variables are treated in 8.2.7, references to entry and generic names (function calls) in 7.5.2. If the reference is a label or a format label, an operand is formed from:

   a) the attribute LABEL
   b) the value representation resulting from the application of the function represent to the denotation of the label or format label (cf. 4.1).
An isub-variable, isub-variables occur only in connection with isub-defining (cf. 8.2.7.2.1); they are associated in the environment directly with an integer value.

8.2.7 Reference to variables

The evaluation of the reference to a variable proceeds in the following steps:

1. 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. 4.2.5).

2. The generation currently associated with the variable is evaluated.

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 (1) also includes checking whether the subscripts are within the range given by the evaluated data attributes of the variable. If a subscript is outside the range, the SUBSCRIPTRANGE condition is called (if enabled). The evaluated data attributes of a variable are obtained:

- for proper variables (i.e. STATIC, AUTOMATIC, or CONTROLLED storage class and parameters) from the data attribute part of the generation of the variable, which for proper variables is immediately accessible, for based and defined variables from the denotation of these variables (cf. 5.4).

Proper variables, defined variables, and based variables differ in step (2), i.e. in the way the generation associated with the variables is obtained. Step 2 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. 7.5.1). The sub-generation resulting from step (3) need not be scalar in the last case.
8.2.7.1 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. The unique name is accessible in the environment part E (see the diagram in 5.3).

No generation exists if the variable has not been allocated.

8.2.7.2 Defined variables

The declaration of a defined variable specifies data attributes and density (PACKED or ALIGNED), optionally a position (an integer which is significant 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.

8.2.7.2.1 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 (2) 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 (1) above) are used to give the values to d isub-variables (those characterized by the integers 1 up to d).

2. The environment taken from the block activation in which the defined variable has been declared is modified by associating in it the names of the isub-variables with their integer values.
(3) The generation associated with the base reference is evaluated, by using the above modified environment for evaluating its subscript expressions.

(4) A consistency check is made between data attributes and density of the elements of the defined array variable, and data attributes and density as given in the above sub-generation. The same conditions must be satisfied as in the correspondence-defined case between attributes of the defined variable and the generation associated with the base reference.

(5) 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 environment under the name of the isub-variable.

**Example:** Let D and B be declared as:

```plaintext
DCL 1 D(2,2) DEFINED B (2SUB,1SUB), 2 X, 2 Y,
1 B(2,2), 2U, 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:

```
1 SUB ... 2
2 SUB ... 1
```

The reference `B(2 SUB, 1 SUB)` is now evaluated in the environment modified in such a way that it shows the association of the isub-variables with their values. 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 sub-generation `gen''` determined by `gen'` and `<2>`.
8.2.7.2.2 Correspondence defining

A defined variable is correspondence defined if no position is specified in the declaration, and if its evaluated data attributes are equal to the data attributes 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. If the data attributes contain strings then the densities of defined variable and based reference must moreover be equal.

The evaluation of the generation associated with the defined variable proceeds in the following way:

1) the base reference is evaluated in the environment of the block activation in which the defined variable was declared, giving the corresponding sub-generation,

2) the data attributes in the data attribute part of this generation are replaced by the evaluated data attributes of the defined variable.

The new data 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 8.2.7.

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. Figure 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.
8.2.7.2.3 Overlay defining

A defined variable is overlay defined if it is a string aggregate and PACKED, and if the base reference is a string aggregate of the same type (BIT or CHARACTER) and also PACKED, and if the condition for correspondence 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 sub-parts 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 \(i\)th up to the \(j\)th element of the base reference, where \(i\) is the integer specified by the position, and \(j - i + 1\) is the number of elements in the defined variable,
(3) A new connected generation is formed using the evaluated data attributes of the defined variable, the density PACKED, 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 k-th element of the defined variable is exactly the storage part associated with the \((1 + k - 1)\)-th element of the base reference (cf. 4,2,7). The new generation is the input to step (3) in the general scheme in 8,2,7.

**Example:** Let \(D\) and \(B\) be declared as

\[
\text{DCL } D(3) \text{ BIT(1) PACKED DEFINED } B \text{ POS(2), } 1 \text{ B PACKED, } 2 \text{ X BIT(2), } 2 \text{ Y BIT(3)};
\]

**Fig. 8.5** symbolically shows the storage associated with \(B\) and its parts, and the corresponding parts of \(D\).

![Diagram of storage associated with B and D](image)

**8.2.7.3 Based variables**

The declaration of a based variable specifies data attributes and density and a pointer reference. Only constant extents are allowed in the data attributes. 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 in the environment of the
block activation in which the based variable was declared. The result is an operand which specifies a pointer value.

(2) A new generation is formed using data attributes and density of the based variable and the pointer value. This generation serves as input to step 3 in the general scheme in 8.2.7.

It is only for certain cases that the resulting generation is a sensible means for referencing storage. Let eda_b be the data 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 data attributes eda_p. If eda_p is scalar, then the value representation associated with the storage part makes sense together with eda_b only if eda_p and eda_b are equal, no relationships between value representations associated with different attributes being defined. If eda_p is non-scalar, then the meaningful parts of the storage part are identified by the storage mapping function map (eda_p, dens_p, i), the storage parts identifiable with the based variable, however are given by map (eda_b, dens_b, i), and again no relationship is defined between the values of the mapping function for different arguments.

There is an exception to the general rule that eda_b and eda_p have to be equal, which is called the left-to-right equivalence rule. This exception is due to a property of the mapping function which is guaranteed by the language. If eda is a structure attribute, then the result of map (eda, dens, i) depends only on the properties of the sub-parts to the left of and including the i-th sub-part. Consequently, if eda_b and eda_p are structure attributes, a reference to the i-th sub-part of the based variable gives defined results if eda_b and eda_p are equal up to the i-th sub-part.
8.3 Infix and Prefix Operations, Conversion, Numeric Pictures

8.3.1 Infix and prefix operations

The part of expression evaluation whose main properties are to be described in this section is the application of infix or prefix operators to their already evaluated, but not yet converted, operands; these operands are objects consisting of a data attribute part and a value representation part, as described in 4.1. The result of the operation is again represented in the form of an operand, and whereas the data attribute part of this result operand is completely defined by the language (except for the implementation-defined maximum precisions associated with arithmetic data attributes), the value represented by the value representation part is generally not. For character string comparison, this has been coped with 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. 8.3.2). For example, for arithmetic infix operators, the common target for conversion of the two operands is the object shown in Fig. 8.6,

```
+ mode + base + scale + prec
```

Fig. 8.6 Target attribute for conversion during arithmetic infix operations

where mode, base, and scale are the higher of the respective characteristics of the two operand attributes; for the arithmetic prefix operators PLUS and MINUS, the target is the object AR-DA (cf. 8.3.2). In the first case, the precision (and where necessary, the scale factor), in the second all characteristics are deduced by the convert-instruction from the incomplete target and the source attribute.
For an infix operation, let $da_1$ and $vr_1$ be the data attribute and value representation of the converted first operand, $da_2$ and $vr_2$ those of the second. The data attribute $da_{res}$ of the result operand is computed as a function of $da_1$ and $da_2$. To obtain the value representation $vr_{res}$ of the result operand, the converted operands are transformed into their values $v_1 = \text{value}(da_1, vr_1)$, $v_2 = \text{value}(da_2, vr_2)$, and a result value $v_{res}$ is computed from $v_1$ and $v_2$ (depending, possibly, also on $da_1$ and $da_2$), then, $vr_{res}$ is obtained as the representation of $v_{res}$ with $da_{res}$.

For arithmetic operators, the first step in the transition from $v_1$ and $v_2$ to $v_{res}$ is a test, whether the operator is applicable; if not, the ERROR or ZDIV condition is raised, otherwise, $v_{res}$ 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 $da_1$, $da_2$. However, the following is postulated:

If $v_1$ and $v_2$ belong to the sets $v$-O-set($da_1$) and $v$-O-set($da_2$) of values which are guaranteed to be exactly representable with $da_1$ and $da_2$, respectively (cf. 4.1.2), then, in case of fixed-point $da_1$ and $da_2$ and an operator which is not division, the result $v_{res}$ will be the exact mathematical result.

**Example:** If $da_1$ is REAL DEC FIXED (3,0), $da_2$ is REAL DEC FIXED (4,1), $v_1$ is 237, $v_2$ is 844.2, and the operator is PLUS, then $v_{res} = 1081.2$, if $v_2$ were 844.25, then $v_{res} = 1081.25$ would not be guaranteed.

Before $v_{res}$ is represented with $da_{res}$, 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 $da_{res}$ the maximum precision associated with $da_{res}$ is used. The following can be derived from the definition of $da_{res}$ and the axioms for $v_{res}$!

If $v_{res}$ is guaranteed to be the exact mathematical result, and if additionally no FIXED OVERFLOW situation arises, then $v_{res}$ is in the set $v$-O-set($da_{res}$), i.e. is guaranteed to be exactly representable with $da_{res}$.

**Example:** In the example given above, if the maximum precision for real decimal fixed attributes is at least 5, then the result attribute $da_{res}$ will be REAL DEC FIXED (5,1), no overflow will occur, and the result value $v_{res} = 1081.2$ will indeed be exactly representable with $da_{res}$.
For comparison operators, the numeric case is treated axiomatically with similar postulates; the character or bit string case, like the string operators anyhow, presents no difficulties; the pointer case again is treated axiomatically, with the following postulates (for the operator EQ; NE is defined as negation of EQ):

a) If the two pointers are the same, then EQ yields true.
b) 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).

8.3.2 Conversion

Conversion is performed by an instruction convert \((\text{da}_t, \text{op})\) which has as arguments the target attribute \(\text{da}_t\) and an operand \(\text{op}\) which is to be converted to this target; the result is the converted operand. The target \(\text{da}_t\) may be incomplete. If so, it is completed (see below). Conversion to a complete target \(\text{da}_t\) falls into three steps:

1) The operand \(\text{op}\) is transformed into a value.
2) The value is converted into a value of the type determined by \(\text{da}_t\).
3) The converted value is represented with \(\text{da}_t\), the result of conversion is the operand whose data attribute is \(\text{da}_t\) and whose value representation is the obtained representation.

The first step and the third are performed by the function \(\text{value}(\text{da}, \text{vr})\) and by the instruction \(\text{test-rep}(\text{da}, \text{v})\), as described in 4.1.2. (The instruction \(\text{test-rep}\) rather than the function represent is necessary, because the SIZE or CONVERSION Condition may be raised).

The second step, called value conversion, 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.
Examples:

(1) If the source attribute $da$ of the operand $op$ and the target attribute $datg$ 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 $da$ and $datg$ are the same, then this operand will be $op$ under certain additional restrictions (cf. 4.1.2).

(2) If $da$ is a bit string attribute and $datg$ 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 $da$ is a numeric picture attribute, $datg$ a character string attribute, then step 1 will yield a numeric value which with the aid of $da$ 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 $da$ and $datg$ are both numeric picture attributes, then the numeric value computed by step 1 will be transformed back into pictured form. Even if $da$ and $datg$ 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 numeric to character conversion, 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. 8.3.3); if it is arithmetic, then again a picture attribute is constructed, though there are some differences to the treatment of the ordinary picture case. Also in numeric to bit conversion, the source attribute is needed. In character to numeric conversion 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 I/O (cf. 9.3.1.). In character to bit conversion, the target attribute is needed, because only as many characters of the source value as necessary are converted (and hence may raise the CONVERSION condition). The two remaining cases, bit to numeric conversion and bit to character conversion, present no problems.
As was said above, the target attribute presented as first argument to the convert-instruction may be incomplete; that means, any component may be specified by *. Examples of incomplete attributes are the following objects AR-DA and STRING-DA (Figs. 8.7.1 and 8.7.2):

![Fig. 8.7.1 The incomplete arithmetic attribute AR-DA](image)

![Fig. 8.7.2 The incomplete string attribute STRING-DA](image)

They can be used to specify "conversion to arithmetic" or "conversion to string" without specifying particular characteristics. Other examples are given in 8.3.1. The completion of the components occupied by * can be done by the convert-instruction with the help of the source attribute.

### 8.3.3 Representation and evaluation of numeric pictures

Only a very brief description of the concepts introduced and used in 8.6 of /6/ 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 12.2.1.2 of /6/ was to make explicit as much as possible of the structure which is needed by the interpreting functions and instructions, without too much 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 (Fig. 8.1. a, b, c):

![Diagram](image)
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 explicit picture attributes; 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.

Examples:

a) For a subfield description 'ZZ.9', the unsuppressed form is '99.9', the explicit form is shown by Fig. 8.9 a (this time strings being presented in their abstract form):

```
   elem(1) elem(2) elem(3) elem(4) elem(5) s-dr-beg s-dr-end s-dr-char
   ┌───────┬───────┬───────┬───────┬───────┬───────┬───────┬───────┐
   │ DOLLAR │ 9-CHAR │ 9-CHAR │ POINT │ 9-CHAR │ 2     │ 3     │ Z-CHAR │
   └───────┴───────┴───────┴───────┴───────┴───────┴───────┴───────┘
```

Fig. 8.9a Subfield of explicit picture attribute
b) For a subfield description '€,999', the unsuppressed form is '€,999', the explicit form is shown by Fig. 8.9 b:

```
 elem(1) elem(2) elem(3) elem(4) elem(5) s-dr-beg s-dr-end s-dr-char
 DOLLAR  COMMA  9-CHAR  9-CHAR  9-CHAR  1  5  DOLLAR
```

Fig. 8.9b 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 (cf. 4.1.2). As an intermediate step in this transformation, the concept of pictured value is used. A pictured value 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.

Example: 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. 8.10):

```
 s-mt-field s-exp-sep s-exp-field
 1-CHAR 2-CHAR 3-CHAR E-CHAR MINUS 3-CHAR
```

Fig. 8.10 Pictured 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 are 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, to its greatest part, 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.
The following abbreviations will be used throughout this chapter:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>attribute</td>
</tr>
<tr>
<td>char</td>
<td>character</td>
</tr>
<tr>
<td>cond</td>
<td>condition</td>
</tr>
<tr>
<td>csa</td>
<td>complete set of file attributes</td>
</tr>
<tr>
<td>ds</td>
<td>dataset</td>
</tr>
<tr>
<td>env</td>
<td>environment</td>
</tr>
<tr>
<td>ev</td>
<td>event, I/O-event</td>
</tr>
<tr>
<td>f</td>
<td>file name</td>
</tr>
<tr>
<td>fd-</td>
<td>file directory-, e.g. fd-element</td>
</tr>
<tr>
<td>fu-</td>
<td>file union directory-, e.g. fu-element</td>
</tr>
<tr>
<td>id</td>
<td>identifier</td>
</tr>
<tr>
<td>int</td>
<td>interpret</td>
</tr>
<tr>
<td>ref</td>
<td>reference</td>
</tr>
<tr>
<td>st</td>
<td>statement type</td>
</tr>
<tr>
<td>t</td>
<td>abstract program text</td>
</tr>
<tr>
<td>tk</td>
<td>take</td>
</tr>
<tr>
<td>u</td>
<td>file union name</td>
</tr>
<tr>
<td>upd</td>
<td>update</td>
</tr>
<tr>
<td>vr</td>
<td>value representation</td>
</tr>
</tbody>
</table>

The totality of actions which is to be described under this heading is of considerable diversity, and it seems advantageous to develop a kind of glossary in order to have a basis for continuation. The explanation of the terms in the glossary is to some extent cyclic; it is hoped that this fact does not override the usefulness of the explanations since the undertaking is not so much a problem of inventing new concepts but of associating terms with concepts (i.e., meaning) where either the terms or the concepts are already known.

**I/O:** anything dealing with input and output (and update); this is a very vague term which is in fact redundant throughout this chapter. However, in conjunction with other terms it has more special meaning, e.g., in I/O statements, record I/O, stream I/O etc. Fig. 9.1 gives a survey on I/O.

**Data transmission:** all actions which correspond to the framed area in Fig. 9.1.

**Elementary transmission:** the actions which are part of data transmission, and are related to the proper transmission of record data or stream data or display data or reply data or comment data.

**Record data:** an object satisfying the predicate is-keyed-data or is-data handled by record I/O (cf. Figs. 9.1 and 9.2). This is usually called "record".

**Stream data:** an elementary object satisfying the predicate is-print-data handled by stream I/O (cf. Figs. 9.1 and 9.3). This is usually called "single character".

9.
<table>
<thead>
<tr>
<th>I/O Type</th>
<th>Standard</th>
<th>System Action</th>
<th>Display of Statement</th>
<th>Display of Standard System Actions on Conditions</th>
<th>Message I/O</th>
<th>Stream I/O</th>
<th>Record I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>check standard</td>
<td>put statement</td>
<td>reply of standard string target</td>
<td>F, E, S</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FD, FU, ES</td>
<td>get statement</td>
<td>get statement on conditions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>read statement</td>
<td>output by:</td>
<td>open or closing output</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>write statement</td>
<td>input by:</td>
<td>read statement on conditions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>name statement</td>
<td>update by:</td>
<td>opened on conditions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>or write statement</td>
<td></td>
<td>terminated or closed output</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>or delete statement</td>
<td></td>
<td>record file</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>system computer</td>
<td>most characteristic</td>
<td>governed by:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>file</td>
<td>statement</td>
<td>record file</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>standard</td>
<td>governed by:</td>
<td>standard file</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>file</td>
<td>record file</td>
<td>stream file</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>standard</td>
<td>standard file</td>
<td>print file</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>file</td>
<td>string source or</td>
<td>message I/O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>standard</td>
<td>display</td>
<td>certain standard</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>file</td>
<td>standard</td>
<td>system on units</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 9.1 Survey on I/O
Inspection of Fig. 9.1 shows that "record I/O" and "I/O governed by a record file" are synonymous but "stream I/O" is wider than "I/O governed by a stream file". Besides of this it is stressed that stream I/O includes special treatment of the standard system print file but there are no peculiarities for a standard system input file.

Remark: As will be seen, the actions of the standard system print file are very similar to the actions of a stream file having the file attributes STREAM, OUTPUT, PRINT. All dissimilarities are restricted to opening and closing.

Explanation: Record data consists of an optional key and a record size (size) and data (vr). The key is either a list of character values (later on referred to simply as characters), e.g. <K-CHAR, 1-CHAR, ..., X-CHAR> or is the null object Ø.

Remark: Record data is available only if access is by a record file; the key is available only if access is by a keyed file.

Fig. 9.2 Record data

Fig. 9.3 Examples for stream data

1) An "Explanation" always conveys additional information which is useful for the first reading of the text.

2) A "Remark" always conveys information which necessarily is not found elsewhere and is oriented towards the more PL/I experienced reader. Remarks could disturb some readers at the very first reading of the text.
Explanation: Stream data are either characters, e.g. A-CHAR, DOLLAR, ..., extra-lingual-characters, e.g., SQUARE, TRIANGLE, ... or the special elementary objects LINE-DELMITER, FILEMARK, PAGE-DELMITER, CARR-RETURN, TABULATOR which could be called control data.

Remark: Stream data is available only if access is done by stream I/O; the control data PAGE-DELMITER, CARR-RETURN, and TABULATOR only have meaning if access is done by a print file. Since extra-lingual-characters are implementation-defined they are not required to be different from control data.

Fig. 9.4 Example for display or reply data

Explanation: Display and reply data consists of a name uniquely identifying the message which is a list of characters, e.g. <K-CHAR, DOLLAR, ..., QUEST>.

Remark: The unique name of a display message is used to associate a specific display message with a specific reply message (namely the reply message having the same name).
display data: an object satisfying the predicate is-named-message handled by the display statement (cf. Figs. 9.1 and 9.4).

reply data: an object satisfying the implementation-defined predicate is-comment handled by certain standard system on-units (cf. Figs. 9.1 and 9.5).

Remark: These comments being put out by certain standard system actions of on-conditions should not be confused with the check standard system action.

The classification of I/O given in Fig. 9.1 is the basis for the structuring of this chapter. Since message I/O is to some extent a special case of record I/O the introduction to message I/O will follow the introduction to record I/O. Record I/O has to be initiated by opening before any data transmitting statements can work on a record file. Consequently opening should be described before record I/O, and since opening is similar for stream and record files the description of opening for all types of files will be the very first subchapter. Chapter 9.2 is record I/O with the description of all statements including the unlock statement (which does not cause data transmission); this is followed by a subchapter describing stream I/O, i.e. the get and put statements including the cases where the statements are governed by a string source or string target, and including the description of the copy action and the check standard system action.

Remark: The copy action is very primitive and is near to the elementary stream output transmission. The check standard system action is higher leveled and is near to a put statement.

Chapter 9.4 is devoted to closing, and message I/O is described in chapter 9.5.

The subchapters which deal with files are in general not organized according to the different I/O statements which can be applied but according to the properties of files.

9.1 Opening

The following abbreviation will be used throughout this chapter:

expl-opt explicit (open) options
The following terms should be understood to have the following meaning:

**opening**: all actions which create (or try to create) a file.

**closing**: all actions which delete (or try to delete) a file.

**file**: the association of a file identifier (or at least of a file name) with a titled dataset in ES obtained by proper opening (and valid until proper closing).

**proper opening**: all actions which create a file; among other actions the fd-element and the fu-element are created.

**improper opening**: all actions which try to create a file but which do not create a file; either the properly opened file already exists or proper opening is not possible because of erroneous situations.

**proper closing**: all actions which delete a file; among other actions the fd-element and the fu-element are deleted.

**improper closing**: all actions which try to delete a file but which do not delete a file; either the file to be closed does not exist or proper closing is not possible because of erroneous situations.

Explanation: Opening and closing is either proper or improper. This should not be confused with explicit or implicit opening which is a classification based on how opening and closing is achieved rather than on what is achieved by opening.

Remark: The totality of files which can be created by explicit opening is wider than the totality of files creatable by implicit opening (linesize, pagesize, dataset label checking, title). In other words any file (including the standard system print file) created by implicit opening could have been created by explicit opening also\(^1\).

**own file**: a term which relates a file with the task in which the file was opened.

**inherited file**: a term which relates a file with the tasks to which the file was passed.

---

\(^1\) the peculiarities of the standard system print file are contained in chapters 9.1.1, 9.1.2 and 9.4.
fd-element: a part of the file directory FD which is associated with one specific file satisfying the predicate is-fd-elem

fu-element: a part of the file union directory FU which is associated with one specific file satisfying the predicate is-fu-elem

Remark: In this context of files it is not necessary to know that fu-elements also exist for I/O with a string source or a string target. In conjunction with files the fu-element satisfies the predicate is-fu-file rather than is-fu-string.

file attribute: one of 13 elementary objects satisfying the predicate is-file-attr; note that the elementary object FILE is not a file attribute.

environment attribute: an implementation defined attribute satisfying the predicate is-io-opt

complete set of file attributes: one of 23 sets of file attributes satisfying the predicate in CSA-LIST.

stream file: a file having in its fu-element a csa containing the file attribute STREAM.

record file: a file having in its fu-element a csa containing the file attribute RECORD.

The meaning of a keyed file, output file, print file, sequential keyed file, etc. should be obvious.

standard system print file: a print file having a special fd-element.

9.1.1 Central part of opening

Any kind of opening as described in the following subchapters ends up with the central part of opening unless actions occur which terminate opening before.

1) cf.2-34(110), 2-35(113), 2-35(114) of /6/
2) the peculiarities of the standard system print file are contained in chapters 9.1.1, 9.1.2 and 9.4.
3) cf. 10-2(2) of /6/
The central part of opening is defined by the instruction

```
open-associate (t, title, new-csa, expl-opt)
```

whose arguments have the following meaning:

- **t** is a piece of abstract program text; in the cases when the instruction is executed as a consequence of implicit opening, \( t \) satisfies one of the predicates `is-delete-st`, `is-file-get`, `is-locate-st`, `is-file-put`, `is-into-read`, `is-set-read`, `is-ignore`, `is-rewrite-st`, `is-unlock-st`, `is-write-st`; if implicit opening is opening of the standard system print file, \( t \) is the null object \( \emptyset \) since in this case the equivalent of a piece of program text does not exist; if the instruction is executed as part of explicit opening of a single file, \( t \) satisfies the predicate `is-open`.

- **title** is the list of characters which serves to identify the titled dataset in ES. For implicit opening this title is derived from \( AT \), for opening of the standard system print file it is the default `SYSPRINT`, and for explicit opening it is derived from the title expression (if specified) otherwise it is the same as for implicit opening.

- **new-csa** is the complete set of file attributes or is the null object \( \emptyset \); the details will be given in the following subchapters.

- **expl-opt** has a meaning for explicit opening only, in all other cases it is the null object \( \emptyset \); it collects the evaluated ident option and the evaluated linesize and pagesize expressions (cf. Fig. 9.5).

![Fig. 9.5](image)

Explanation: `ds-label` is either a generation or a pseudo-generation (to receive a dataset label) or a list of characters (to supply the dataset label) or the null object \( \emptyset \).
The instruction open-associate has four alternatives, the last of which is the only one making proper opening.

The first alternative is taken when at least one of the following cases occurs:

- the argument new-csa is the null object instead of a reasonable csa which indicates that a conflict in the attributes has been detected previously,
- there is no (or more than one) titled dataset in ES which is identified by the second argument title,
- the dataset description accessed by the title is improper,
- the outer dataset accessed by the title is improper,
- the outer dataset cannot be deciphered appropriately, i.e., it is impossible to map it into a proper inner dataset being compatible with new-csa.

The consequence of this alternative is the execution of invalid-open (t) which either returns the truth value T (for explicit opening) or calls the undefined-file condition (for implicit opening) or results in an error (for the standard system print file).

The second alternative is taken whenever there already exists a file which is an own and/or direct file. In this case the value F is returned.

The third alternative is the erroneous case of an inherited, non-direct file.

The fourth alternative is proper opening of a file (which may be the standard system print file) which results in the execution of the instruction open (u, t, title, new-csa, expl-opt), where u is a free unique name which will be used as a selector to the fu-element being created by the instruction open-fu.

Explanation: The creation of the fu-element takes place only if the titled dataset will be accessed by a single file or by an arbitrary number of direct files. In this case the fu-element will be initialized, otherwise execution results in an error.

1) any values T or F returned by the instruction open-associate only have meaning with explicit opening where they denote subsequent raising (or not raising) of the undefined-file condition, respectively.
Initialization for every file consists in making an entry for the complete set of attributes new-csa (under s-csa of fu-element), for the title as derived from AT (under s-at-title of fu-element), for the environment attribute from AT (under s-io-env of fu-element), for the file name (under s-f of fu-element), and for the title which was the second argument in open-associate (under s-title of fu-element). These five components of fu-element will not be changed during the existence of the file.

Remark: Together with the linesize and pagesize entries (see below) they are the only components which remain unchanged.

Explanation: The two titles of the fu-element are used as follows. The title derived from AT is nothing else than the file identifier in the form of a list of characters which supplies the condition built-in function onfile with the proper value. The other title serves to identify the titled dataset in ES.

Depending on new-csa some other components of fu-element will also be initialized: the position (under s-position of fu-element) for all non-direct files, the set of all attached I/O-events (under s-io-ev of fu-element) for all record files, the current column (under s-column of fu-element) for all stream files, the current line (under s-line) and the pagesize (under s-pagesize) for all print files, and the linesize (under s-linesize) for all stream output files.

Besides creating an fu-element, proper opening also creates an fd-element. In contrast to the fu-element no components of the fd-element will be changed throughout the existence of the file.

Creation of the fd-element is done in two steps. The first step open-fd-1(u, t, new-csa) has three alternatives which are "direct" opening, "indirect" opening, and the erroneous case of "indirect" opening. Indirect opening is opening of the standard system print file if, it occurs in connection with a put statement. In this case an fd-element is created which has two parts (cf. Fig. 9.6), the part being accessible through the file identifier SYSPRINT with the file name fs being linked with the proper entry for the standard system print file which itself provides the link u to the fu-element.

1) some remarks on the current file are contained in chapter 9.1.2.2
2) the terms direct and indirect opening serve only for local explanatory purposes in connection with opening of the standard system print file.
Remark: In any stage of the computation FD either has one or no fd-element which characterizes the standard system print file.

Direct opening is opening either of a file which is not the standard system print file or of the standard system print file by means of a copy action or check standard system action (cf. Fig. 9.1). In other words, direct opening occurs if the file name is different from $f_s$ or if it is the default file name $s$-stand-print.

The erroneous case of indirect opening means that the standard system print file was opened directly, and subsequently indirect opening should be performed.

Remark: The two appearances of the standard system print file are based ultimately on the fact that a copy action or check standard system action does not provide a contextual declaration of the file identifier SYSPRINT with external scope.

Processing of dataset labels is done after creation of the fu- and fd-element. Both the header and trailer dataset labels are immediately accessible in the titled dataset, and it should be noted that the mapping functions cipher and decipher are not used for dataset labels.

9.1.2 Implicit opening

This subchapter describes any opening which is not executed by an open statement, i.e. implicit opening, as it occurs with all I/O statements naming a file by a file identifier, or naming the standard system print file.

9.1.2.1 Opening with a file identifier

The abstract program text $t$ is as described in chapter 9.1.1 but it never satisfies the predicates $\text{is-open}$ or $\text{is-Q}$.
Opening is performed in three major steps. The first is merely a test as to whether the identifier occurring in t really is a file identifier (test made by the instruction `verify-file-name`). The second step (which occurs only if the first step has not been erroneous) is the execution of the instruction `open-associate` as described in chapter 9.1.1, the third argument of which (new-csa) is the set of attributes derived from the type of the statement t (default-attr-set(t)) united with the set of file attributes from AT (the declared attributes) and again united with `{PRINT}` in the case of "indirect" opening of the standard system print file.

The third step, `verify-impl-open (t)`, is the consistency test required between all options occurring in t and the csa of the fu-element. If opening in step two was improper and no file exists the error condition has to be called, otherwise either there is consistency or execution results in an error.

If opening was proper, the file union name is returned.

9.1.2.2 Opening without a file identifier

This is "direct" opening of the standard system print file which mainly consists in the execution of actual opening by the instruction `open-associate` having as first argument the null object $\varnothing$ (since no statement text exists), the title SYSPRINT as the second argument (in the form of a character list), the csa `{STREAM, OUTPUT, PRINT}` as third argument, and the null object $\varnothing$ as fourth argument.

Remark: The concept of a current file, the file identifier of which must be available for the condition built-in function onfile, requires that execution of every statement be preceded by updating the component s-curr-file of the condition-built-in-function part of CS. For non-I/O-statements having no component for a file identifier this updating results in setting the component of CS to the null object $\varnothing$. If a conversion condition call is executed, the current file identifier is taken from that component of CS.
9.1.3 Explicit opening

This subchapter describes the interpretation of the open statement.

Since the structure of an open statement (cf. Fig. 9.7) essentially is a list of "minor open statements" (the first element of the list is depicted in some detail), and since the structure of such a minor open statement is similar to the data transmitting I/O-statements, the execution of a minor open statement is similar to implicit opening with a file identifier.

Differences with respect to implicit opening consist in the evaluation of the options a minor open statement can have (selected by s-ident, s-title, s-linesize, s-pagewidth), deduction of the file attributes not from a statement type but from the explicitly specified attributes (e.g., file-attr-set), interpretation of the values T or F returned by any execution of a minor open statement (cf. chapter 9.1.1). This value is used to raise the undefined file conditions (if any) in the order in which the minor open statements appear in the open statement.
9.2 Record I/O

Key to the abbreviations used in this chapter

<table>
<thead>
<tr>
<th>c</th>
<th>opt</th>
<th>sequ</th>
</tr>
</thead>
<tbody>
<tr>
<td>control</td>
<td>statement</td>
<td>sequential</td>
</tr>
<tr>
<td>te</td>
<td>options</td>
<td>tv</td>
</tr>
<tr>
<td>task or event specification</td>
<td>unique task or event name</td>
<td>task variable</td>
</tr>
</tbody>
</table>

This subchapter is devoted to the interpretation of the data transmitting statements write, rewrite, locate, delete, read into, read set, and read ignore (the last three being the syntactically separable forms of the read statement), and to the non-data transmitting unlock statement.

9.2.1 Classification of data transmitting statements

For all data transmitting statements the sequence of actions is essentially the same and is described in the following.

1) Definition of the range of the statement, i.e., there is a test whether the into reference (in a read into statement) or from reference (in a write or rewrite statement) refers to a variable, and whether the identifier (in a locate statement) refers to a based variable.

2) Evaluation of all options, i.e., generations of into, from, keyto, set, and event references are established; expressions are evaluated in the key expression (optionally in the rewrite, delete, read into, and read set statements), the keyfrom expression (optionally in the write and locate statements), and the ignore expression (read ignore statement).

---

1) This subchapter is based on work done by V. Kudielka.
Remark: It is assumed that the translator has resolved the default case already.

Since evaluation of any expressions involving keys is by the instruction `convert` (with target attribute `CHAR-DA` which is the data attribute of a character string of unspecified length) and evaluation of the ignore expression is by the instruction `convert` (with target attribute `BINTG-DA` which is the data attribute of a binary fixed point integer of default precision) condition raising follows the rules for conversion.

3) Implicit opening of the file and test for compatibility of the file and the statement. As described for opening, the statement with all its options is checked against the csa. ¹ This guarantees, for example, that a read into statement is valid only on a non-output file, which must not have a key reference if the file is non-sequential, and which must not have a no-lock option if the file is non-exclusive. Implicit opening can be erroneous or can cause the undefined file or error conditions to be called.

4) For sequential files and with the read statement only, the endfile condition can occur immediately, if the endfile status has been reached already by a previous statement.

5A) For statements with an event reference the I/O-event is attached (cf. chapter 9.2.3 on I/O-events).

5B) (mutually exclusive with 5A)

For buffered files (cf. chapter 9.2.2.3) and if a buffer has been allocated already by a locate, read into, or read set statement (where the buffer is noted under `s-bufferpointer`, and if the statement specifies a key for keyed buffered files the key is also entered under `s-key` of the fu-element), the contents of the buffer (for keyed buffered files with the key) is transmitted to the dataset. The key-, record-, and transmit conditions may be called. (For details on the elementary transmission see part 7.)

¹) is-consistent-record-file-1, cf. 10-15(36) of /6/
is-consistent-record-file-2, cf. 10-15(37) of /6/.
6A) For exclusive files the test is made as to whether the record data is locked or unlocked and locking is performed if necessary (cf. chapter 9.2.4).

6B) (mutually exclusive with 6A)

For buffered files the old buffer is omitted (i.e. freed and the noted buffer and key if any deleted from the fu-element), a new one (cf. par. 5B) is allocated, and the pointer is assigned to the set reference in the case of a read set statement.

7) Elementary transmission

7.1) Since for any data transmission the (whole) inner dataset is needed, the first instruction executed is tk-dataset (u,key). As outlined in chapter 4.3 the file union name u gives access to the fu-element in which all information is available to decipher the titled dataset in FS. It should be kept in mind that the instruction returns the (whole) inner dataset, and that the optional key (specified for keyed files only) being the second argument of the instruction is not used to select specific record data from the inner dataset.

Remark: The whole inner dataset is needed because for keyed files the tests on keys (cf. par. 7.2) have to be made; for (different) direct files working on the same titled dataset (i.e. having different fu-elements perhaps with a slightly different cas, environment attribute, etc.) one file may change the dataset in parts which are not known by the other file.

7.2) Test for multiple occurrences of keys (on output) or for non-existent keys (on input). The key condition can be called.

7.3) For sequential files the position is updated (s-position of the fu-element). It may be altered by +1 (or -1 for files with the backwards attribute) or by any other positive integer (read ignore) or by a direct access, i.e. by a key expression in a read into or read set statement. The endfile condition can be called.

7.4A) For input the record and transmit conditions may occur.¹)

¹) cf. chapter 9.2.3 if an I/O-event was attached in par. 5A.
7.4B) (Mutually exclusive with 7.4A)

For output the new inner dataset is formed and also the record and transmit conditions may occur. 1)

7.5) The actual data transmission is performed, i.e. on input the record data is assigned to storage or buffer, on output the (whole) inner dataset is handled by \texttt{upd-dataset(u,key,ids)}. As explained in par. 7.1 above the fu-element contains all necessary information to cipher the titled dataset in ES.

7.6) For buffered input the second step, from buffer to storage is performed.

Explanation: Whenever input and output has been used in par. 7.2, 7.4, and 7.5 this should also include update files. This is the reason why the more restricted terms input file and output file were not used.

7.7) The check condition may be raised for the key to reference in a read into or read set statement, for the set reference in a read set or locate statement, or for the into reference in a read into statement. 1)

8) For exclusive files unlocking of the record data may be performed. 1)

9.2.2 Actual data transmission

For all types of record files this subchapter describes how a specific record data is extracted from an inner dataset (on input), and how the new inner dataset is built up (on output); this refers to parts of par. 7.4 and 7.5 of chapter 9.2.1.

9.2.2.1 Direct files

In actual output data transmission the new inner dataset is formed by the instruction \texttt{from-direct}. The arguments are a key (already in the form of a list of characters), the pointer taken from the generation of the from reference, and the (whole) inner dataset. This inner dataset is guaranteed to be a set (by opening), and this set is united with the record data which can be derived by the arguments of the instruction and storage S.

1) cf. chapter 9.2.3 if an I/O-event was attached in par. 5A.
Remark: On direct update files the instruction \textit{from-direct} executed in the interpretation of a rewrite statement is preceded by the deletion of the record data to be rewritten.

Actual input transmission is performed by the instruction \textit{direct-into} which has a key, the pointer taken from the generation of the \textit{into} reference, and an inner dataset as arguments. The key designates the record data to be taken from the inner dataset, the value representation of which should be assigned to the part of storage \( S \) determined by the pointer.

9.2.2.2 Unbuffered files

On output, the record data which is the element of the list indicated by \( s\)-position of the \( fu\)-element is to be added (or replaced) by new record data which is obtained by the pointer of the generation of the from reference and storage \( S \). This is executed by the instruction \textit{from-sequ} which has the position, the pointer, the optional key, and the inner dataset as arguments and returns the new inner dataset.

Actual input transmission in all cases consists of the assignment of the value representation of the record data designated by the position to the part of storage \( S \) determined by the pointer taken from the generation of the \textit{into} reference. This is done by the instruction \textit{sequ-into} having the position from the \( fu\)-element, the pointer, and the inner dataset as arguments. However, if the keyto reference is present in a read into or read set statement the key of the record data available under s-key (cf. Fig. 9.2) will also be assigned to the part of storage \( S \) designated by the pointer of the generation of the keyto reference. This assignment is performed by the instruction \textit{sequ-keyto} which may cause a key condition call.

9.2.2.3 Buffered files

Since the buffer which is itself a part of storage \( S \) and the key (for a keyed buffered file) must be accessible by the file the reference to the buffer is made possible by a pointer which is noted under \( s\)-bufferpointer of the \( fu\)-element, and the key is accessible directly under \( s\)-key of the \( fu\)-element (cf. point 5B in chapter 9.2.1).

The allocation of the buffer in a read into or read set statement is done only by giving the size of the record data accessible by \( s\)-size (cf. Fig. 9.2) since
no data attributes and density are known. This is done by the instruction
**buffer-allocate** which returns the pointer identifying the allocation. In a locate
statement the buffer allocation has no special problems and can be made using the
instruction **based-allocate**.

In all cases where the buffer is no longer needed the buffer is freed and the
entries s-bufferpointer and s-key (if any) will be deleted. This occurs in the
locate, read into, read set, and delete statements and in closing buffered files.

The actual output transmission which is performed by the instruction
**buffer-sequ** is very similar to unbuffered transmission. The instruction has the
file union name and the inner dataset as arguments and returns the new inner data-
set (taking the position, the bufferpointer and the key - if any - from the fu-
element).

Actual input transmission is carried out in two steps. The first step is the
assignment of the record data designated by the position, to the part of S de-
signated by the bufferpointer which is executed by the instruction **sequ-buffer**
having the file union name and the inner dataset as arguments. In the second step
(executed only in a read into statement) the buffer designated by the bufferpointer of
the fu-element is assigned to the part of S designated by the pointer from the
generation of the into reference; this is made by the instruction **buffer-into**. In-
stead of the instruction **sequ-keyto** used for unbuffered files, for buffered files
the assignment of the key is also made in two steps if the keyto reference is
present. The first step is the instruction **note-key** (entering the key of the
record data into the fu-element), and the second step is done by **key-buffer-keyto**
(assigning the key from the fu-element to the storage denoted by the keyto refer-
ence).

9.2.3 **I/O-events**

The attaching of an I/O-event is similar for all record I/O statements and
for each specific case where an event reference may occur. For this reason the ac-
tions to be performed as an I/O-event are gathered into a single instruction which
is executed as a consequence of the first alternative of the instruction
**int-io-event**; this instruction is executed whenever a statement occurs which can
have an event reference, and if the file is either direct or unbuffered. If there
is no event reference the data transmission instruction will be executed by
**io-transmission** which is part of the second alternative of the instruction **int-io-event**.
Otherwise the I/O-event is attached, and the essential effect is to enter the
piece of control represented by the instruction *io-transmission* into the parallel task and I/O-event part PA as an autonomous parallel action. Thus, the instruction *io-transmission* is executed either as an I/O-event or directly, i.e. in the control of the task in which the statement occurs.

The detailed actions of the instruction *attach-io-event(type,u,opt,te, ten)* will be explained in the sequel.

The argument type is one of nine elementary objects which are not used by *attach-io-event* but by *io-transmission* to discriminate between the different kinds of data transmission; the argument u is necessary to retrieve the fu-element; the argument opt (cf. Fig. 9.8) collects all evaluated options of the statement; the argument te is established by *int-io-event* and consists of the unique name of the current task (under s-ten) and the generation of the event reference (under s-ev); the last argument ten is a (new) unique name.

![Diagram](image)

**Fig. 9.8 The object opt**

Explanation: opt collects all evaluated options a statement can have. The figure does not show that some or all evaluated options can be the null object \(\emptyset\), and that some options are mutually exclusive. A similar object is used in explicit opening (expl-opt).

First a test is made as to whether the event reference shares storage with an active task or an event variable. Then the assignment to the event reference is made setting the variable to normal and incomplete; in the task and event specification TE (of the current task) the unique name ten of the I/O-event attached is united with the set of the unique names of previous I/O-events attached in this task \((\text{io-ev-set}_\text{task} \text{ in Fig. 9.9})\); in the file union directory FU the unique name ten is also entered (under \(\text{s-io-ev(FU)}\)); in PA the I/O-event is attached under its unique name ten.
Fig. 9.9 gives an example for PA after attaching an I/O-event with the unique name ten in a task with unique name ten\textsubscript{task}. In the attached I/O-event, \(\text{gen}_{\text{ev}}\) is the generation of the event and \(\text{c}_{\text{trans}}\) is the appropriate alternative of the instruction \texttt{io-transmission} followed by the instruction \texttt{activate-tasks-1} which informs all tasks that the data transmission has terminated. Under the selectors \(\text{s-cond-list}\) and \(\text{s-check-list}\) the arguments for the I/O-condition calls and check condition calls will be entered during or before execution of \(\text{c}_{\text{trans}}\) in order to raise the conditions appropriately at the wait for the I/O-event. Similar entries are made under \(\text{s-unlock}\) if \(\text{c}_{\text{trans}}\) accesses an exclusive file (cf. chapter 9.2.4). In this case record data stays locked as long as the wait for the I/O-event is executed. The data attributes of the references gathered under \(\text{s-check-list}\cdot\text{s-te}\cdot\text{ten}(\text{PA})\) can be derived by means of the current environment stored together with the references.
Any task can attach an arbitrary number of tasks and I/O-events but no I/O-event can attach a task or I/O-event.

An I/O-event is called semi-complete if the control $c_{\text{trans}}$ has become the null object. As well as raising any check and I/O conditions, the wait statement performs unlocking, sets the event complete (through $\text{gen}_{\text{ev}}$) and deletes the I/O-event from $\text{PA}$. Unlocking does not occur in the case when the I/O-event has been attached by a read into statement without nolock option.

9.2.4 Locking and unlocking

Locking is not a property of the dataset but of the file which must be an exclusive file. For this case the fu-element contains the so-called locking information (cf. chapter 4.3) which associates every locked key with the unique task name of the task in which the record data designated by the key was locked.

Any read into statement without the nolock option and any delete or rewrite statement accessing an exclusive file waits as long as the key is unlocked (through execution of the instruction wait-for-unlock) and then locks the key itself.

Explanation: Since the locking mechanism works only for one file but for all tasks accessing this file it makes no difference whether the record data is said to be locked or the key identifying the record data.

Closing a file deletes the fu-element which has the effect of unlocking at closing. At task termination where closing of all files owned by this task is performed, in addition all keys in inherited files locked by this task have to become unlocked which is done by the instruction unlock-taskend.

9.2.5 Remarks on record I/O statements

The classification of data transmitting statements given in chapter 9.2.1 is considered to be the basis for the interpretation. Many references to other subchapters given there supply all the details necessary for one specific statement.

An exception is the unlock statement for which only the paragraphs 2 and 3 of chapter 9.2.1 apply. Details are given in chapter 9.2.4.
9.3 Stream I/O

Key to the abbreviations:

| cl | character list | gen | generation |
| contr | controlled | i | integer |
| d | data | nl | nested list |
| eval | evaluate | ons | onsource |
| expr | expression | ps | pseudo |
| fo | format | spec | specification |
| fol | format list | tr | transmission (mode) |
| f u- | file union directory |

This subchapter contains the description of the get and put statements including the copy action and the check standard system action.

9.3.1 Get statement

9.3.1.1 Source specification

A get statement either specifies a file identifier (Fig. 9.10) or it specifies a string source by a reference to a character string variable (Fig. 9.11).

![Fig. 9.10 'Get statement with a file identifier'](image)

The execution of a get statement with a file identifier consists of implicit opening, testing of the end-of-file status of the file,\(^1\) initialization of the count and onsource components\(^2\) of the fu-element, evaluation of the optional expression specifying skipping of lines, execution of skipping, and the interpretation of the data specification list. During the execution of the statement the file accessed by the file identifier acts as source of data. Actually data is supplied by the dataset or from fu-ons.

---

1) fu-endf in Fig. 4.14
2) fu-count and fu-ons in Fig. 4.16
Whenever the get statement specifies a string source, a fu-element will be created before the interpretation of the data specification list, in order to unify further treatment of a file and of a string source by giving in both cases a file union name as an argument.

The comparison of Fig. 4.16 with Fig. 9.12 shows that fu-fol, fu-comma, and fu-ons also have meaning with a file. Only the component fu-data is specific for a string source: it is a copy of the source character string.

Remarks: At the beginning of any get statement the component fu-ons is set to the null object $\mathcal{Q}$. This value remains unchanged throughout the execution of the statement unless a conversion on-unit is executed because of incorrect data, and in the on-unit an assignment to the onsource pseudo-variable is made. Upon normal return from the on-unit the value assigned to the pseudo-variable will be copied into fu-ons and acts (as long as it contains characters) as the actual source of data. There is no difference between a file and a string source in the treatment of fu-ons; however, there is a difference with respect to the type of the get statement because in edit-directed transmission fu-ons will not be used.

The component fu-comma only has meaning in a get statement performing list-directed transmission. The component is set to the truth value T whenever the delimiting comma between two data-fields is transmitted or if a situation occurs which is equivalent (e.g. the initialization of the fu-element of a string source).
9.3.1.2 Data specification

For the sake of generality the structuring of the data specification list was changed in order to be the same for get and put statements.

Any element of the data specification list either is of list-directed, data-directed or edit-directed type (Fig. 9.13).

Any item-list in Fig. 9.13 either has expressions or controlled items as its elements.

A controlled item as shown in Fig. 9.14 has similar structuring and meaning as a controlled do group; in the case of a controlled item, the subject of iteration is an item list again (the component selected by s-do-list), in the case of a controlled do group it is a statement list. For this reason the iteration mechanism of a controlled do group can be used until the point where the interpretation of the statement list would occur.

The instruction int-contr-do which executes the iteration has an additional argument (the I/O-link) which conveys the information needed in order to resume interpretation of the item list of the controlled item.

---

1) The necessity of a data specification list instead of a single data specification arises only in the edit-directed case.
The components of the I/O-link in Fig. 9.15 are a statement type, the file union name (in order to access the fu-element for further use), and a component which characterizes the transmission mode (tr). The transmission mode specifies in the cases of list- or data-directed transmission whether the copy action has to occur; in the edit-directed case it is more complicated.

Besides giving the edit-directed type and the indication for the copy action, the whole format list (format-list in Fig. 9.16) from the respective element of the data specification list and an integer (integer1) are conveyed. The whole format list is needed for the case when wrap around occurs, the integer which is always greater than zero is the format list index needed to select the element from fu-fol (cf. Fig. 4.16) which is itself a list and represents the format list which is worked upon. The format list index is unique for any statement working on the fu-element. The evaluation of format lists is described in chapter 9.3.1.3.

If the first element of item-list in Fig. 9.13 is a reference, the data attributes of this reference are evaluated and are used to govern the expansion into the (scalar) generations of this reference. To each of these generations the operands are assigned which originate from the data fields (cf. chapter 9.3.1.4).

In the case of data-directed transmission the data fields themselves have to contain the reference in a special format. As for the constants in list-directed transmission, the reference is translated into the equivalent reference (as it would occur in abstract program text); for this reference the generation can be evaluated, appropriately tested and the assignment can be made.
9.3.1.3 Format list

As for the previous subchapter, most of the description of format lists applies to the get and put statements.

Although the format list occurs in concrete and abstract text at the same level as the data list (cf. Fig. 9.13: item-list, and format-list) the interpretation of the format list is mainly governed by the data list. One of the exceptions is the around of the format list.

The evaluation (or expansion) of the format list is done by the instruction `eval-format (u, spec-fol, fo-i)`. The parameters are used as follows:

- **u** is the file union name of the fu-element,
- **spec-fol** is the format list from the transmission mode (i.e. format-list in Fig. 9.16),
- **fo-i** is the format list index from the transmission mode (i.e. integer in Fig. 9.16).

The instruction has four alternatives which have the following meaning:

The first alternative is taken if the format list denoted by **fo-i** in **fu-fol** is the empty list (i.e. the wrap around case) or does not exist (i.e. the first access to **fu-fol** by a statement); in both cases **fu-fol** is replaced by **spec-fol** and the instruction is entered again with the same arguments.

The second alternative is taken if the first element of **fu-fol** is an iterated format; in this case the expression which specifies the repetition factor is evaluated and the format itself is concatenated (as often as the repetition factor specifies) with the tail of **fu-fol**; this resulting format list then replaces **fu-fol** and the instruction is entered again with the same arguments.

The third alternative is the remote format; upon evaluation of the value of the reference (of the format label) and upon testing of the value the remote format list replaces the remote format in **fu-fol** and the instruction is entered again with the same arguments.

The fourth alternative is taken for a data or control format; all expressions occurring in the format are evaluated and the evaluated format is returned as the result of the instruction.

---

1) cf. Figs. 9.12 and 4.16
9.3.1.4 Data fields

A list- or data-directed get statement transmits stream data as long as the data field is complete and correct or the transmission is temporarily suspended at the first stream data which is recognized as incorrect. In both cases it is essential that defined actions follow the terminated or suspended transmission.

Correct data fields can be generated by a set of context free production rules and if the grammar defined by the rules is unambiguous the generation history also defines a unique parsing. A systematic transcription of the rules yields a set of predicates of abstract syntax which describe a correct data field e.g. as a nested list of characters, i.e. as an object.

Let cl₁ be a list of characters, and it is asked whether cl₁ is a correct data field. If this question is rephrased it reads as follows: Is there an object which satisfies the transcribed predicates of abstract syntax (i.e. is there a nested list nl₁) which upon unnesting¹) equals cl₁?

Example: With the transcribed rule for a decimal-integer-constant which can be depicted as

```
< >
digit
```

or

```
[elem(1)
decimal-integer-constant
elem(2)]
```

where "digit" stands for 0 or 1 or ... 9 and "<>" is the empty list, the answer on the question as to whether e.g. cl₁ = <4,9,3> is a decimal-integer-constant in the above sense is yes, since there is indeed a nested equivalent nl₁ = <<<>,4>,9>,3>, i.e. unnest-cl(nl₁) = cl₁, and nl₁ can be depicted as

```
1) unnest-cl is a one-place function whose domain is the set of arbitrarily nested lists of characters and whose range is the set of lists of characters. The nested argument list and the resulting list define the same sequence of characters.
which satisfies the transcribed rule.

Let \( cl_1 \) be a list of characters: it is the first wrong character in the data field that is asked for. If this question is rephrased it reads as follows:

1) If the data field is correct (see above) the question does not apply.
2) (Incorrect data field)
   Take the list which is composed of the leftmost \( i \) characters of \( cl_1 \) and ask for \( i = 1, 2, \ldots \) whether there exists a correct continuation. As long as the answer is yes the index \( i \) is incremented until the answer is no. This is the first wrong character.

Let \( cl_1 \) be a list of characters: it is the parsing of the data field that is searched for.

This parsing essentially is this nested list \( nl_1 \) which satisfies the transcribed predicate and where \( cl_1 = \text{unnest-cl (} nl_1 \text{)} \).

9.3.1.5 Copy action

The transmission mode (cf. Figs. 9.15 and 9.16) specifies whether the copy action has to occur or not. The copy action effectively takes place during elementary data transmission, i.e. for every single stream data which is a character and which is taken from the dataset. For this single character the elementary stream output transmission is executed with the file union name of the standard system print file.
9.3.2 Put statement

9.3.2.1 Target specification

A put statement either specifies a file identifier or it specifies a string target by a reference to a character string variable or pseudo-variable. Same as for a string source a fu-element will be created before the interpretation of the data specification list.

An fu-element for a string target (Fig. 9.17) has in its component fu-gen the generation or pseudo-generation of the reference which is evaluated during initialization of the put statement. The component fu-data serves as intermediary target, and before completion of the statement the contents of fu-data is assigned to the generation fu-gen.

Most parts of the chapters 9.3.1.2 and 9.2.1.3 are applicable to the put statement also.

9.3.2.2 Check standard system action

The check standard system action performs opening of the standard system print file and then executes the actions for a data-directed data specification consisting only of one reference. In order to do so the interpretation for the data-directed data specification must not fail for references declared as entry or label and for proper variables of other than arithmetic, string or picture type.
9.4 Closing

The usual effect of closing is to delete a file which is the reverse of what is normally done through opening. The terms used in connection with closing are defined in chapter 9.1 because of the similarity with opening.

Closing proceeds in four steps described in the sequel:

1) (Only if closing is by a close statement, i.e. explicit closing)
   The actions are decomposed into actions for a "minor close statement"\(^1\) which occur in unspecified order, and which are preceded by a test as to whether the identifier really identifies a file.

2) (Only if closing is by a minor close statement)
   For an own file the ident option is evaluated and after processing of dataset labels proper closing is made. The access to an inherited file by a minor close statement is erroneous.

3) (Proper closing)
   This step also occurs at task termination for all own files.
   
   3A) (Only for a direct or unbuffered file)
       Any I/O-events still attached are deleted from PA and the assignment to abnormal and complete is made.
   
   3B) (Only for a buffered file)
       The buffer (eventually with a key) is transferred to the dataset.

   3C) (Only for the standard system print file if opened indirectly)
       The entry of FD selected by s-stand-print is deleted.

4) The fd-element and the fu-element is deleted.

9.5 Message I/O

Data transmission to and from message storage \(M\) as it occurs e.g. in the interpretation of the display statement is described in this subchapter.

---

1) This term should be obvious from the similarity between the abstract text of open and close statements (cf. chapter 9.1.3).
9.5.1 Message storage M

A global state component M serves to accumulate all messages in the sequence they are put out by the PL/I machine (in the case of display messages or comments) or put in from somewhere outside the machine (in the case of reply messages).

![Diagram of message storage](image)

Fig. 9.18

The structure of message storage is given in Fig. 9.18, examples for messages can be looked up in Fig. 9.4.

During the computation the lists selected by s-display and s-comment are incremented only, and always on their tails. Both lists are no more inspected by the PL/I machine, i.e. have no further meaning for the computation.

The component selected by s-reply is never changed from the PL/I-machine; changes are assumed to occur from outside the PL/I machine which inspects the reply messages only.

9.5.2 Display statement

The abstract text of the display statement, as demonstrated by two examples in Fig. 9.19, consists of two optional references, one being the variable to receive the reply message, the other being the event variable the presence of which implies the presence of the reply reference.
The first action of any display statement is to evaluate the display expression in the current environment, convert it to a character string of unspecified length, and append it together with a new unique name to the tail of the list of display messages in message storage. Then the generations of the reply and event references (if any) are evaluated, and the interpretation of the display statement will be completed if there is no reply reference.

If there is a reply reference of type character string but no event reference, an instruction \texttt{int-reply} will be executed which is of wait type (i.e. effectively prevents the task in which it is executed to continue until the proper reply message is put into the list of reply messages of message storage from outside the PL/I machine). In this case waiting does no more take place but the message is converted from a list of characters into a character string operand and is assigned to the generation of the reply reference and the check condition is called.

If there is an event reference, an I/O-event is attached (cf. chapter 9.2.3), and the instruction \texttt{int-reply} is given into the control of the I/O-event.

\section*{9.5.3 Output of comments}

Comments being put out by certain standard system actions of on-conditions are treated as display messages. Since comments do not require to be paired with any kind of answer, they need not be identified by a unique name.
10. CONDITIONS

The following abbreviations will be used throughout this chapter:

- `cap`: condition action part
- `cond`: condition
- `CS`: condition status
- `cond-bif-part`: condition builtin function part
- `D`: dump
- `dyn-pref-part`: dynamic prefix part
- `EI`: epilogue information
- `env`: environment
- `id`: identifier
- `intg-val`: integer value
- `n`: unique name
- `st`: statement
- `st-type`: statement type

10.1 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.

![Diagram of Condition State CS](image)

The block-prefix-part (selected by s-bpp), and the statement prefix-part (selected by s-spp), control the condition enabling status for blocks or procedures and statements, respectively.

The condition action part (cap) 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.

10.2 Enabling and Disabling for Conditions

At the beginning of program interpretation, a system defined 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 and
procedures, or of statements.

To ensure unambiguity of reference to conditions which are characterized
by identifiers, a dynamic condition selector (cond-dyn-sel, cf. 2-3o(100) of
/6/), is created by connecting the unique name of the identifier with the
identifier and a simple selector.

Example 1:

(CHECK(A,B)); BEGIN;

A = B;
END;

dynamic condition selectors: A = n₁ • s-check
B = n₂ • s-check

where n₁ and n₂ are the unique names associated with A and B.

Where no unique name is needed, for example with the conversion
condition, the dynamic condition selector consists only of the simple
selector, e.g. s-conv.

Thus a dynamic interpretation of condition prefixes is ensured. The
dynamic condition selector is also used to connect a condition with the
proper condition action (cf. chapter 10.4).

As condition prefixes heading a begin block or a procedure statement
have the scopes of the respective blocks, they are interpreted at block
entry (cf. int-block 5-3(1) of /6/) or at procedure entry (cf.
int-proc-body 5-50(90) of /6/), respectively. The updating of the condition
enabling status of a block or procedure is done by merging the condition pre-
fixes 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. 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.

10.3 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 \( CS \).

![Fig. 10.2: Condition Action](image)

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.

For a check condition, the condition action is established for every base element for a structure reference, and for every alternative element for a cell reference.

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 specified, is inherited to all descendents 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.

10.4 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 immediately. The actual condition call is interpreted in the same way as when the condition is raised by interrupt, by the instruction call-cond-1.

The activation of the various conditions is described in the various places in the interpreter, where they can occur (see for example raising of the check condition with the assignment statement; cf. 7-3(1) of /6/).

Special actions have to be performed for the check and the conversion condition, before the general interpretation of the condition call by call-cond-1. The check condition, raised with a reference list, has to be expanded (cf. chapter 10.3), and then the call is executed to every element of the expanded reference list. The ordering of elements in the list is relevant. For the conversion condition, a dummy has to be created for the onsource pseudo-variable, before the actual condition call may be interpreted.

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 condition is enabled or not. This information has to be tested before raising. Only the raising of the underflow condition and the check 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.

The condition state is now inspected for an appropriate condition action for the specific condition, which is done through the dynamic condition selector. 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 (cf. int-cond 9-13(24) of /6/). The current state is stacked, and a new block activation with the environment taken from the condition action 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 (cf. 5.2.2 of /6/). When this is finished, the block activation is terminated by the instruction epilogue (cf. 5.1.7 of /6/). The stacked state is reinstalled and the next instruction is executed. In some cases this may be the error instruction which finishes the interpretation.

For the interpretation of the standard system action, the language 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 for the standard system action by the endpage condition, the check condition and the error condition. The standard system action for the finish condition results in no action.

### 10.5 Condition Builtin Function Status

The information needed to interpret condition builtin functions is kept in the condition builtin function part of CS (cf. 10.1). This part consists of one component for each condition builtin function, and of four additional components.

![Fig. 10.3: Condition-bif-part](image-url)
The values of the components of the condition builtin functions are updated every time a condition action different from SYSTEM is interpreted (cf. 10.4). The auxiliary components, with the exception of the part selected by s-ten, get their values directly in various places of the interpreter. All the other values are entered into a special argument constructed at the point of the condition call and the argument is then passed to the instruction update-cbf (cf. 8-8(15) of /6/). 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 entry component of the condition-bif-part of CS; the onfile value, in the case of the conversion condition, is taken from the curr-file component. The value of oncode is defined by an implementation defined function. The third auxiliary component characterizing the statement type is used to distinguish conditions raised by interrupt from conditions raised by a signal statement. The component selected by s-ten is needed to provide proper completion of i/o-events after a GOTO out of an on-unit called during a wait statement.

The components of the condition builtin function part may then be used by the condition builtin functions. 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 $\varnothing$, the functions return standard values as described for the individual builtin functions.