Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

Contribution to the design of a database design workbench
Logical record acces evaluation

Delcourt, Bruno; Moentack, Luc

Award date:
1984

Awarding institution:
University of Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 04. May. 2026

Facultés Universitaire Notre-Dame de la Paix (Namur)

Institut d'Informatique

CONTRIBUTION TO THE DESIGN

OF A DATABASE DESIGN

WORKBENCH

LOGICAL RECORD ACCESS EVALUATION

Mémoire présenté par

Bruno DELCOURT,

Luc MOENTACK,

en vue de l'obtention
du titre de
licencié et maître en informatique

Année académique 1983 - 1984

Acknowledgmnents.

First of all we would like to thanks professor Hainaut who accepted to conduct this thesis.

We are greatly indebted to professor Teichroew and professor Kang for providing us with all facilities for this work, and to professor Teorey who gave us the opportunity to attempt his class on database design, at the University of Michigan.

We are also grateful to Mr. Deville. His valuable hints were fully appreciate all along this year.

TABLE OF CONTENTS

INTRODUCTION.

1 . INTRODUCTION TO INFORMATION PROCESSING SYSTEMS.

2 . THE GAM MODEL.

2 .1 . Introduction. 2 .2

2 .2 . The GAM objects. 2 .3

2 .3 . The data designation language. 2 .8

2 .4 . Integrity constraints. 2 .12

3 . THE ADL DEFINITION.

3 .1 . Constituents and features of ADL. 3 .3

3 .2 . Basic symbols, names, numbers, strings. 3 .5

3 .3 . Variables. 3 .9

3 .4 . Functions. 3 .11

3 .5 . General expression. 3 .12

3 .6 . Statements. 3 .27

3 .7 . Declaration. 3 .37

4 . THE STATISTICAL MODEL.

4 .1 . Introduction. 4 .2

4 .2 . Presentation of the statistical model. 4 .2

4 .3 . E-R schema of the statistical model. 4 .4

5 . GENERAL ARCHITECTURE OF A LOGICAL DB DESIGN TOOL.

5 .1 . Introduction. 5 .2

5 .2 . Conceptual schema of the database. 5 .3

5 .3 . Description of the steps. 5 .11

5 .4 . An example. 5 .14

6. INTRODUCTION TO ISLDM/SEM.

6 .1 . Life cycle support sytems. 6.2

6 .2 . ISLDM. 6.5

6 .3 . SEM 6.6

7 . DESCRIPTION OF THE IMPLEMENTATION OF SDLA.

7 .1 . Introduction. 7.2

7 .2 . The Meta Model. 7.3

7 .3 . Description of the SDLA language. 7.5

8 . INTRODUCTION TO LEX AND YACC.

8 .1 . Introduction 8.2

8 .2 . YACC 8.4

8 .3 . LEX 8.7

8 .4 . Cooperation between LEX and YACC. 8.9

9 .DESIGN AND IMPLEMENTATION OF THE ADL ANALYSER.

9 .1 . Introduction 9.2

9 .2 . Tree structure. 9.2

9 .3 . Symbol table structure. 9.9

9 .4 . Lexical analysis. 9.15

9 .5 . Syntactical analysis. 9.16

9 .6 . Semantics analysis. 9.24

10 . THE LOGICAL RECORD ACCESS EVALUATOR.

10 .1 . Introduction. 10.2

10 .2 . Goal of the evaluation. 10.3

10 .3 . Example and problem. 10.4

10 .4 . Evaluation method. 10.11

10 .5 . Criticism. 10.34

CONCLUSION

INTRODUCTION

Large databases today contain hundreds of thousands of records. In such environments, considerations of performances are particularly important. The performance of database systems however, depends on a large number of interdependent factors (data model, data placement on device, database contents, access algorithms ...).

Experience has shown that performance of existing database systems may be significantly affected by the efficiency of data manipulation algorithms implemented in application software. The number of "logical record access" (LRA) to the database appears, at the logical design layer, as one of the best criteria to measure the efficiency of access algorithms.

Several LRA evaluators have been designed in various works. They can be classified with respect to two criteria :

- the complexity of the database access algorithms.
- the accuracy of the statistical description of the database contents required for their computations.

The less sophisticated tools on this criteria scale evaluate simple queries and assume the uniformity and independence of the data distributions, whereas the more complex ones use general distributions and data correlation to measure the performances of particular DBMS data manipulation algorithms.

In this work we present the design of an LRA evaluator which is more precise than less sophisticated evaluators, and requires less information than sophisticated ones. Besides, we suggest an overall framework for predicting the effects, in terms of logical record accesses, of logical data base design decisions. The fundamental components of this database design workbench are the data structure model, and the algorithm description language. In order to be able to assess performances, we provide an LRA evaluator for analysis of arbitrary complex data manipulation algorithms supported by a statistical description facility.

The first part of the work concerns the environment of the LRA evaluator, namely i) the data structure model, ii) the algorithm description language, and iii) the statistical model. The purpose of this first part is two-fold. We hope to acquaint the reader with the basic concepts of these models, and we show how we implemented them.

INTRODUCTION

The first chapter introduces the three main layers of a life cycle support system. Chapter two presents the concepts of the data structure model, namely the Generalized Access Model (GAM). We insist on the GAM ability to represent a wide range of data structures. Chapter three gives a complete description of the algorithm description language (ADL). It focuses on the facilities that the database access constructs offer. Chapter four explains the statistical description model. Careful attention has been given to the choice of the distribution functions. These models are not independent from each other. Both the statistical description and algorithms are related to a database schema. Therefore, we need to integrate these models. Chapter five presents the conceptual schema of the 'global' model. Moreover, we suggest a database design strategy which use the 'global' model.

An overview of the ISDOS software (University of Michigan), which will support the logical database design workbench, is presented in chapter six. Chapter seven discusses the implementation of the global model, using ISDOS software.

In order to be able to analyse ADL algorithms we implemented an ADL parser. Chapters eight and nine discuss the implementation of the ADL parser, using a parser generator (YACC & LEX).

The second part of the work proposes an evaluation method and its criticism (chapter ten). Stress has been laid on the necessity of a stepwize evaluation method.

Chapter 1: INTRODUCTION TO INFORMATION PROCESSING SYSTEMS.

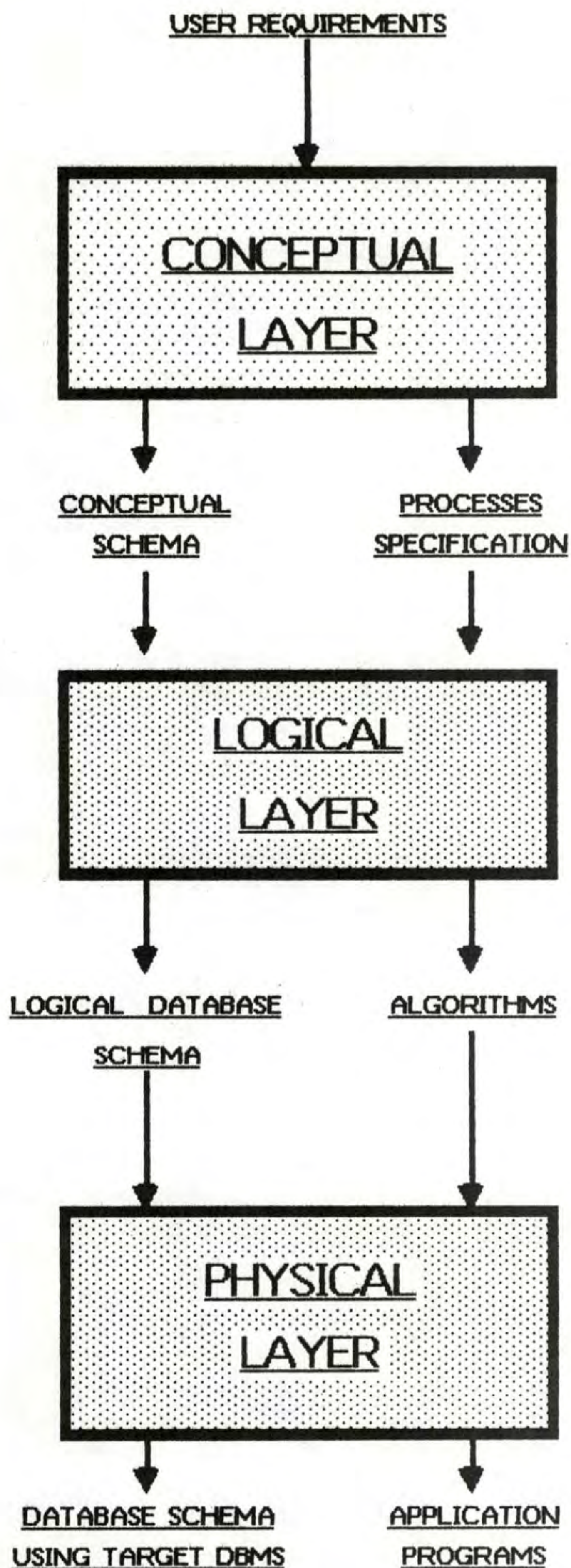


FIG.1.1

Generally speaking, the development of an information processing system is divided into three main layers : the conceptual layer, the logical layer, and the physical layer. Each layer is supported by its own set of models. At the logical layer we will use the GAM model to describe the data structure, the statistical model to describe the statistical properties of the data, and the ADL language to describe processes.

Each layer can be viewed as a black box. Figure 1.1 shows the dataflow between the layers.

A particular methodology which is developped for the conceptual layer, is given in [BOD.83].

An example of a physical design methodology is given in [TEO.83].

In this work we suggest an overall framework for database design, at the logical layer.

Chapter 2: THE GENERALISED ACCESS MODEL

1. Introduction

The Generalised Access Model (GAM) is a "data organisation" model. It allows one to describe any Database Management System's (DBMS) or File Management System's (FMS) "data organisation". By "data organisation", we intend a data structure and the set of actions that can be performed on it.

Although DBMS appear very different, research has shown that they rely always on the same concepts. The GAM is a complete collection of these widely used concepts.

It offers a general and simple way to express the access structures necessary to an application set without considering any particular DBMS or FMS. This is a key property because we do not want to be already bounded to a DBMS at the logical design step.

We will give :

- a description of the GAM objects.
- an introduction to the data designation language.
- some integrity constraints.

Examples and graphical representation of each concepts will be included. More detailled description can be found in [HAI.84A].

2. The GAM objects

The model objects are RECORDS, RECORD TYPES, DATA ITEM VALUES, DATA ITEMS, ACCESS PATHS, ACCESS PATH TYPES, FILES, ACCESS KEYS, ORDERS.

2.1. RECORD and RECORD TYPE

A RECORD is an information set. It is the logical communication means between program and database. It can be created, deleted or accessed by a program.

The RECORD TYPE defines the common properties of a RECORD set. A RECORD belongs to one and only one RECORD TYPE. At a given time 0, 1 or more RECORDS belong to a RECORD TYPE.

Graphical representation :

A RECORD TYPE is represented by a named box :

2.2. DATA ITEM VALUE and DATA ITEM

We call "domain" a data type like string, real number, integer number. A domain element is a data type value. A domain can be associated to a record type; this association is called DATA ITEM. An identifying name is given to each DATA ITEM of a record type. A domain element associated to a record, occurence of the record type, is called a DATA ITEM VALUE.

Unlike records, DATA ITEM VALUES can not be created or deleted, they exist by themselves. One attach a DATA ITEM VALUE to a record or detach it from a record. Access to the DATA ITEM VALUE is possible by accessing the record to which it is attached.

DATA ITEMS are divided into four major classes :

- "Elementary" or "Decomposable" DATA ITEM. A DATA ITEM VALUE of an elementary DATA ITEM is not meaningfully decomposable into smaller information pieces. A DATA ITEM VALUE of a decomposable DATA ITEM is a list of meaningfull domain element. A list component is abusively called a DATA ITEM. It is recursively elementary or decomposable.

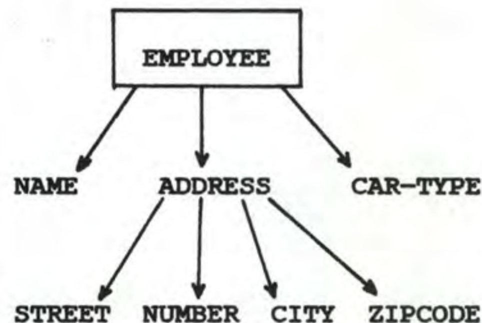
- "Simple" or "Repetitive" DATA ITEM. A DATA ITEM is simple if one and only one corresponding DATA ITEM VALUE can be

attached to each record. A DATA ITEM is repetitive if more than one correponding DATA ITEM VALUE may be attached to each record.

The concepts of simple/repetitive and elementary/decomposable are independent.

Graphical representation :

A DATA ITEM is represented by its name and an arrow linking the representation of the associated record type to it. The arrow pointing to the DATA ITEM shows that it is accessible from the record type.



NAME is a simple DATA ITEM. ADDRESS is a decomposable DATA ITEM. CAR-TYPE is a repetitive DATA ITEM.

In section 4 we will enhance the graphical representation in order to be able to graphically distinguish simple and repetitive DATA ITEMS.

2.3. ACCESS PATH and ACCESS PATH TYPE

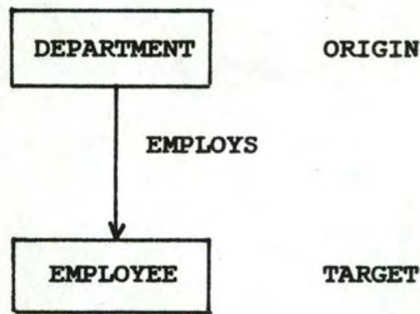
An ACCESS PATH links a record (called ORIGIN) to zero, one or more records (called TARGETS). Targets are accessible from their origin.

Each ACCESS PATH belongs to one and only one ACCESS PATH TYPE which defines the common properties of an ACCESS PATH set. An ACCESS PATH TYPE is caracterized by a non mandatory name and by its origin and target record type names.

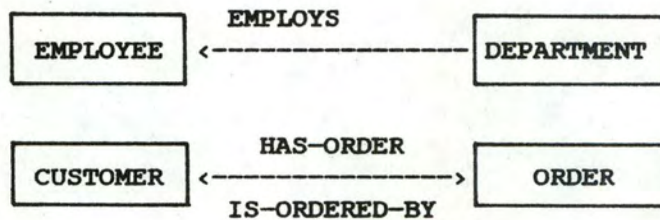
An ACCESS PATH TYPE P1 is the INVERSE of an ACCESS PATH TYPE P2 if : given T target of an ACCESS PATH P1 which has O as origin, an ACCESS PATH P2 linking T to O exits.

Graphical representation :

- An ACCESS PATH TYPE is represented by a named single headed arrow pointing to the target record type.



- Two inverse ACCESS PATH TYPES are represented by a double headed arrow labelled with one or two names. Meaningful names are usually given to ACCESS PATHS.



2.4. FILE

A FILE is a dynamic collection of records. A record belongs to one and only one FILE. Many FILES may collect the same record type and many record types may be stored in one FILE.

2.5. DATABASE

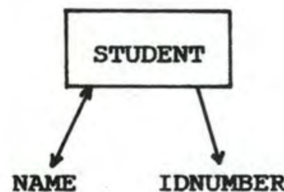
A database is a shared collection of interrelated data designed to meet the needs of multiple types of users. Logically, it is the record collection of a set of files and the data structures associated.

2.6. ACCESS KEY

An ACCESS KEY is an access mechanism. It gives sequential access to a selected record set. The selection criterion is : "the records having a data item or data items group with a given value". The ACCESS KEY is usually identified to the data item or data items group.

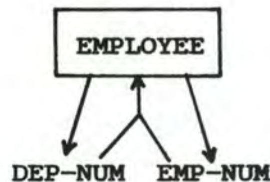
Graphical representation :

- single data item ACCESS KEY are represented by a double headed arrow joining the data item representation and the record type representation.



NAME is an ACCESS KEY to the record type STUDENT.

- ACCESS KEYS based on data item groups are represented by join arrows pointing from the data items representations to the record type representation.



The data items group DEP-NUM, EMP-NUM is an ACCESS KEY to EMPLOYEE

2.7. ORDER OF A SEQUENCE OF RECORDS

Many times we have made explicit or implicit reference to the concept of records sequence. Databases, files, access path targets and sets of records selected by an access key are records sequences. These sequences are ordered.

We distinguish to order types :

- Simple order : if the records sequence is composed by records belonging to one and only one record type.
- Global order : if the records sequence is composed by records belonging to many record types.

Example :

- **Random order or no order**
- **Order of insertion in the sequence (first in first out, last in first out)**
- **Sorted order based on a data item or data items group.**

3. The DATA DESIGNATION LANGUAGE

As the graphical representation,the Data Designation Language (DDL) is not part of the Generalised Access Model. Simplicity in the expression of integrity constraints was the main reason to introduce the DDL in this chapter. The language is a subset of the Access algorithm Description Language (ADL see chapter 3). We give the syntax and semantic in a non formal way. More rigourous description is to be founded in [HAI.84A].

3.1. NOTATIONS

Although the GAM allows empty access path type names, we will, for practical reasons, restrict the model by assigning a name to each access path type. Record types, access path types and data items will be represented by their name.

3.2. SETS

We consider record sets and data item sets. Sets may be labelled by a set name.

Syntax :

<name><condition>

- <name> is a record type name, data item name or set name.
- <condition> is the expression of a condition (see section 3.3).

Semantic :

It represents the occurences (records, data item values, or set elements) which satisfy the condition.

Examples :

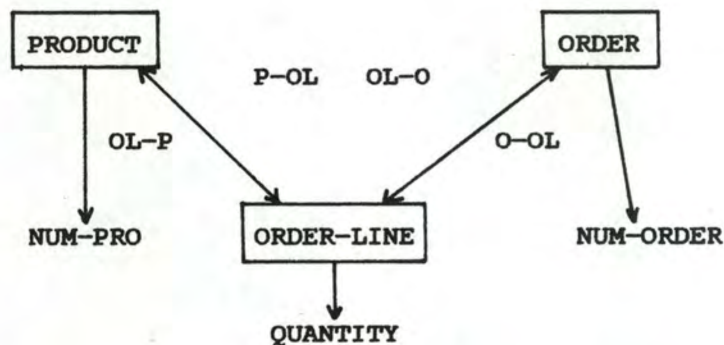


fig 2.1

Fig 2.1 is the graphical representation of a database schema expressed in the GAM model.

- PRODUCT : represents all the records PRODUCT contained in the database (empty condition).

- ORDER(:NUM-ORDER = 783) : represents the records ORDER which have NUM-ORDER equal to 783 (see section 3.3)

- QUANTITY > 50 : represents the values of the QUANTITY data item domain higher than 50 (see section 3.3).

3.3. CONDITIONS

A condition is a simple condition or a boolean expression of simple conditions. A condition makes it possible to select a subset of a set (called the qualified set). Suppose the type (record type or data item) of the qualified set elements is named "set-type".

In this section, we distinguish two types of simple conditions : "cardinal relationship condition" and "belonging condition".

- Cardinal relationship criterion :

Selection of qualified set elements is made with respect to the number and properties of the elements related to them.

Syntax :

(<relationship>:<cardinal><related-set>)

- <related-set> is the designation expression of a set. Suppose the type of its elements is named "related-set-type".

- <relationship> is the name of the relationship between the types "set-type" and "related-set-type". Relationship between record types and data items have empty names.

- <cardinal> is the designation of a set of integers.

We consider only integer intervals :

[1:5] represents the integers 1,2,3,4,5.
[0:*] represents all the integers.
[1:1] represents the integer 1.
An empty <cardinal> stands for [1:*].

Semantic :

An element of the qualified set is selected if it is related to I elements of the "related-set" with I belonging to <cardinal>.

Examples :

If we consider the database schema of fig 2.1 :

- PRODUCT(P-OL:ORDER-LINE) represents the records PRODUCT related by P-OL to at least one ORDER-LINE.

- NUM-PRO(:PRODUCT(P-OL:ORDER-LINE)) represents the values of the data item NUM-PRO associated to the above PRODUCT records.

- ORDER(O-OL:[5..10] ORDER-LINE) represents the records ORDER related by O-OL to more than 4 and less than 11 ORDER-LINE records.

- Belonging condition :

The condition relates to the belonging or non belonging of a qualified set element to a collection of compatible type elements. This condition is usually used to select data item values.

Syntax :

(<rel-op><S>)

and

<rel-op><S>

- <rel-op> is a relational operator (= , <> , <= , >= , < , > , in , not-in).

- <S> is the designation of a set. The set elements must be of compatible types (see chapter 3 for restrictions in the use of <rel-op>).

Semantic :

The condition is true if the qualified set element (considered as a set) and <S> satisfy the relation identified by <rel-op>.

Examples :

If we consider the database schema of fig 2.1 :

- NUM-PRO < 100 represents the values of the data item NUM-PRO lower than 100.

- ORDER-LINE(:QUANTITY = QUANTITY(:OL)) represents the subset of ORDER-LINE records which have their quantity data item with the same value as the one of the ORDER-LINE record represented by the set label OL. This is a combination of belonging condition :

  (= QUANTITY(:OL))

  and cardinal relation conditions :

  (QUANTITY(:OL) , ORDER-LINE(:QUANTITY = QUANTITY(:OL)).

3.4. FUNCTIONS

In some cases, the expression of a condition requires the use of functions giving, for example, the size of a set or the rank of an element in a sequence. We will give the definition of some widely used functions.

- SIZE(E) gives the size of the set designated by the expression E.

- MIN(E) (MAX(E)) gives the lower (higher) value of the set designated by the expression E. This set must be non empty.

- ORD(e,E,O) gives the rank of the element e in the set E. O specifies the order type (see section 2.7).

4. INTEGRITY CONSTRAINTS

The goal of the Generalised Access Model (GAM) is to give a DBMS independant expression of the data and access structures necessary to an application set. These structures derives from the conceptual specifications [HAI.80A].

The GAM basic concepts introduced in section 2 lead to a too general representation of the conceptual data structures which have to satisfy integrity constraints.

We need a set of rules corresponding to the conceptual integrity constraints. These rules will also be called "integrity constraints". New integrity constraints may appear due to modifications in the database schema. A typical example is the equivalence of redundant data structures introduced in order to improve performances or security of the system.

Although, the wide range of integrity constraints will be expressed with the Data Designation Language (see section 3), some frequently used constraints like functional class, existence constraint and identifier, become Generalised Access Model concepts.

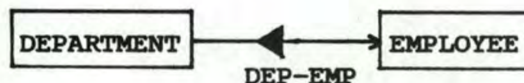
4.1. Functional class

The functional class property applies to association types. Suppose R is an association type between two object types A and B. The functional class relates to the maximum number of B (or A) objects associated to one A (or B) object. Functional class is an oriented concept, when speaking about the functional class of association R, one has to specify its origin and target object types.

One-to-many (1-N) association type

The functional class of the association type R (origin A, target B) is one-to-many if the maximum number of A objects associated by R to one B object is one.

- formally : For each b of type B, size(A(R:b)) <= 1
- example and graphical representation :

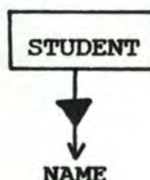


DEP-EMP (origin DEPARTMENT, target EMPLOYEE) is a 1-N association type. An employee belongs to only one department. A triangle is added to the representation of the association type in order to represent a one-to-many functional class. The triangle points to the "one" side.

Many-to-one (N-1) association type

The functional class of the association type R (origin A, target B) is many-to-one if the maximum number of B objects associated to one A object is one.

- formally : For each a of type A, size(B(R:a)) <= 1
- example and graphical representation :



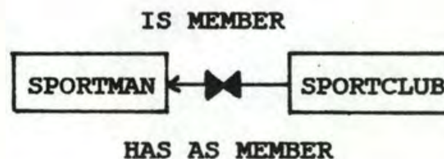
The data item association NAME is a many-to-one association type between STUDENT (origin) and NAME (target). A student has only one name. A triangle is added to the representation of the association type in order to represent a many-to-one functional class. The triangle points to the

"one" side.

Many-to-many (N-M) association type

If R (origin A, target B) is a many-to-many association type, there is no constraint on the maximum number of A objects (respectively B objects) associated by R to one B object (respectively A object).

- example and graphical representation :



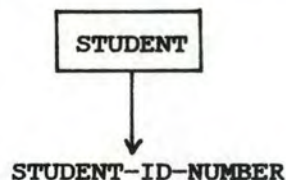
A SPORTMAN can be member of many SPORTCLUB and a SPORTCLUB has many members. A double triangle is added to the graphical representation of the association type in order to represent a many-to-many functional class.

One-to-one (1-1) association type

The functional class of the association type R (origin A, target B) is one-to-one if the maximum number of B objects associated to one A object is one, and if the maximum number of A objects associated to one B object is one.

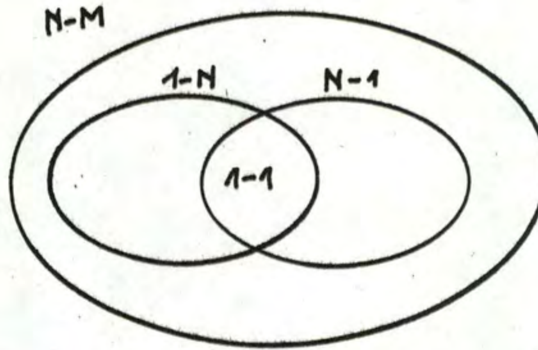
- formally : For each a of type A, size(B(R:a)) <= 1
  For each b of type B, size(A(R:b)) <= 1

- example and graphical representation



The data item association STUDENT-ID-NUMBER is a one-to-one association type between STUDENT and STUDENT-ID-NUMBER. A STUDENT has only one STUDENT-ID-NUMBER and there is only one STUDENT corresponding to one STUDENT-ID-NUMBER.
A one-to-one association type is represented by a non modified single or double headed arrow.

Remq :



Many-to-one and one-to-many association types are particular cases of many-to-many association type. A one-to-one association type is a one-to-many and a many-to-one association type. This implies that properties defined on one class of associations are still valid in the included classes.

4.2. Identifier

We consider two kinds of identifiers, simple identifier and multiple identifier, depending upon wether an object is identified by another object or by a group of objects.

- Simple identifier :

  A target object of a one-to-many or one-to-one association type identifies the corresponding origin object.

- formally : B is "identifier" of A via the R (origin A, target B) association if :

  for each b of type B, size(A(R:b)) <= 1

- example and graphical representation :

  DEPARTMENT

  DEP-NUMBER

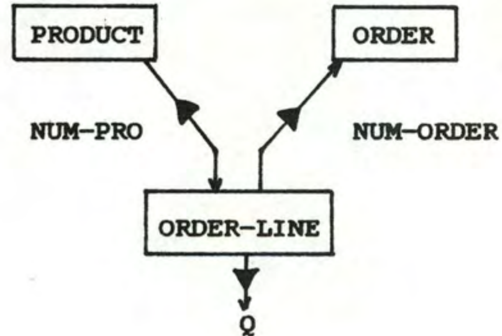
  The data item DEP-NUMBER is identifier of the record type DEPARTMENT. An identifier is represented by the "many" side of a one-to-many association type.

- Multiple identifier

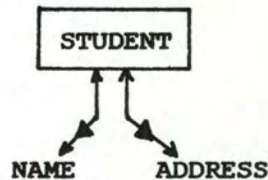
  Suppose that a group of object types B1,B2,B3,... is associated by R1,R2,R3,... to an object type A. The group is a multiple identifier if each tuple b1,b2,b3... identifies an object A.

- formally : for each b1 of type B1, b2 of type B2, b3 of type B3 ..., size(A((R1:b1) and (R2:b2) and (R3:b3) ...)) <= 1

- example and graphical representation :



An ORDER-LINE is identified by a PRODUCT record and an ORDER record. The group of record types PRODUCT,ORDER is a multiple identifier of the record type ORDER-LINE.



The record type STUDENT is identified by the data item group NAME,ADDRESS.

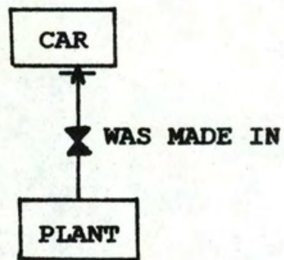
A multiple identifier is represented by parallel association representation on the side of the identified object type.

4.3. Existence constraint

This constraint imposes the association between a record type and an object type at any given time. In other words, an occurence of the record type concerned by the existence constraint has to be associated by a specified association type to an occurence of another object type.

- formally : if the existence constraint concernes the record type A associated by R to the object type B.
  A(R:B) = A

- example and graphical representation :



There is an existence constraint on the association "WAS MADE IN" between CAR and PLANT. A car has always be made in a plant.
The existence constraint is represented by a dash on the representation of the association type on the side of the constrained record.

Chapter 3: ADL DEFINITION

SUMMARY.

This chapter contains a defining description of ADL. This is a language suitable for expressing a large class of database manipulation algorithms.

In the first section, a survey of the basic constituents and features of the language is given, and the formal notation, by which the syntactic structure is defined, is explained.

The second section lists all the basic symbols, and syntactic units known as names, numbers and strings are defined.

Then, the basic components of a general expression are discussed. These are variables (section 3), and functions (section 4).

The fifth section explains the rules for forming expressions and the meaning of these expressions. This section dicusses database expressions, which constitute a powerful feature of ADL.

The sixth section describes the operational units of the language, known as statements.

In the seventh section, the units known as declarations, serving for defining permanent properties of the units entering into a process described in the language, are defined.

The report ends with an alphabetical index definition.

1. INTRODUCTION.

1.1. Structure of the language.

The purpose of the Access Algorithms Description Language is to describe, not only conventional computational processes, but also database processes. The basic concepts used for the description of calculating rules is the well known arithmetic expression. On the other hand we will use the concept of DB_object_set to describe database accesses.

To show the flow of processes, certain statements are used so that may describe, for examples, alternative, or iterative processes.

Statements are supported by declarations which are not themselves computing instructions but informs the translator of the existence and certain properties of objects appearing in statements, such as the class of numbers taken on as value by a variable, the dimension of an array, the reference to the database description schema.

Therefore, a program consists of two major parts. The first part is the declaration part. The second part, namely, the statement part consists of a set of statements enclosed between the statements BEGIN and END.

1.2. Formalism for syntactic description.

The syntax will be described with the aid of metalinguistic formulae. Their interpretation is best explained by an example

<ab>::=(| [| <ab>(| <ab><d>

Sequence of characters enclosed in the brackets < > represents metalinguistic variables whose values are sequences of symbols. The marks ::= and | (the latter with the meaning of OR) are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequence denoted. Thus the formula above gives a recursive rule for the formation of values of the variable <ab>. It indicates that <ab> may have the value (or [or that given some legitimate value of <ab>, another may be formed by following it by the character (or by following it with some value of the variable <d>. If the values of <d> are decimal digits, some values of ab are :

[(((1(37(
(12345(
(((
[86

In order to facilitate the study, the symbols used for distinguishing the metalinguistic variables have been choosen to be words describing approximately the nature of the corresponding variable. Where words which have appeared in this manner are used elsewhere in the text they will refer to the corresponding syntactic definition.

2. BASIC SYMBOLS, NAMES, NUMBERS, AND STRINGS.

2.1. Basic symbols.

The ADL language is build up from the following basic symbols :

<basic_symbol> ::= <letter>|<digit>|<logical_value>|<delimiter>

2.1.1. Letters.

<letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|

A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

Letters do not have individual meaning. They are used for forming names and strings.

2.1.2. Digits.

<digit> ::= 0|1|2|3|4|5|6|7|8|9

Digits are used for forming numbers, identifiers and strings.

2.1.3. Logical values.

<logical_value> ::= true|false|TRUE|FALSE

The logical values have a fixed obvious meaning.

2.1.4. Delimiters.

```
<delimiter> ::= <operator>|<separator>|<bracket>
               |<declarator>

<operator> ::= <arithmetic_operator>|<test_operator>
              |<logical_operator>|<sequential_operator>
              |<set_belonging_operator>
              |<relationship_operator>

<arithmetic_operator> ::= +|-|*|/|^

<test_operator> ::=  <|>|=<|>=|<>|=

<logical_operator> ::= and|or|not

<sequential_operator> ::= for|while|if|create|modify
                         |detach|attach|transfer|return

<set_belonging_operator> ::= in|not in

<relationship_operator> ::=  (:

<separator> ::=  ,|.|;|:=|:|do|then|else|order|from|via
                |endif|endfor|endwhile

<bracket> ::=  (|)|[|]|{|}|begin|end

<declarator> ::= pas|procs|funcs|var|array|of
                |structure|list of|record|real|integer
                |boolean|string
```

Delimiters have a fixed meaning which for the most part is obvious or else will be given at the appropriate place in the sequel.

Typographical features such as blank space or change to carriage return have no significance in the language, except between keywords and names. They may, however, be used freely for facilitating reading. For the purpose of including text among the symbols of a program the following "comment" convention hold :

```
/* <any sequence of characters not containing '*/'> */
```

2.2. Names.

2.2.1. Syntax.

```
<name> ::= <name> <letter>
          |<name> <digit>
          |<letter>
```

2.2.2. Examples.

```
i
kilo
student
```

2.2.3. Semantics.

Names have no inherent meaning, but serve for the identification of simple variables, arrays, functions, algorithms, schema. The same name cannot be used to denote two different quantities.

2.3. Numbers.

2.3.1. Syntax.

```
<unsigned_integer> ::= <unsigned_integer> <digit>
                      |<digit>

<decimal_number> ::= <unsigned_integer>.<unsigned_integer>
                   | .<unsigned_integer>
                   | <unsigned_integer> .

<sign> ::= +|-

<exponent_part> ::=  E <sign> <unsigned_integer>
                     E <unsigned_integer>

<unsigned_real> ::= <decimal_number> <exponent_part>
                  |<decimal_number>
```

2.3.2. Examples.

```
34
34.
.34
34.34E-34
```

2.3.3. Semantics.

Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of ten.

2.3.4. Types.

Integers are of type ´integer´. All other numbers are of type ´real´.

2.4. Strings.

2.4.1. Syntax.

<string> ::= ´ <any sequence of basics symbols not containing ´> ´

2.4.2. Examples.

´hello the world´

2.4.3. Semantics

In order to enable the language to handle arbitrary sequence of symbols, the string quotes ´ is introduced. Strings are used as parameters of functions, and may be stored and be processed in ´string´ variables.

3. VARIABLES

3.1. Syntax

```
<variable> ::=              <hierac_variable>

                            |<non_hierac_variable>

<non_hierac_variable> ::= <name>

                            | <subscripted_variable>


<subscripted_variable> ::= <name> [ <subscript_list> ]

<subscript_list> ::=        <general_expression> , <subscript_list>

                            | <general_expression>

<hierac_variable> ::=       <field_selection> . <variable>

<field_selection> ::=       <non_hierac_variable>
```

3.2. Examples.

```
CUSTOMER
ADDRESS.CITY
TAB[I*J,9]
ADRESS[67].ZIPCODE
```

3.3. Semantics.

A variable is a designation given to a memory cell where a single value, or a set of values can be kept. These values may be used in expressions for forming other values and may be changed at will by means of assignment statements (section 6.2). The type of the value(s) of a particular variable is defined in the declaration for the variable itself , or for the corresponding array identifier , or structure identifier, or list identifier , or record identifier. (section 7.2).

3.3.1. Subscripts.

Subscripted variables designate values wich are components of multidimensional arrays. Each general expression of the subscript list occupies one subscription position of the subscripted variable, and is call a subscript. The complete list of subscripts is enclosed in the subscripts brackets. The array component referred to by a subscript variable is specify by the actual numerical value of its subscripts.

Each subscript positions acts like a variable of type integer, and the evaluation of the subscript is understood to be equivalent to an assignement to this fictious variable. The value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array.

3.3.2. Hierachy.

Hierac. variables designate values wich are components of structured variables the field selected in the structure and is called a field selection. A field selection is always a non hierac variable designation. Two consecutive field selection are separated by a period. The left most field selection is the name of the structured variable. The type of the value is the the type of the variable indicated by the right most field selection.

4. FUNCTION DESIGNATION.

4.1. Syntax.

```
<func>  ::=      <name> <parameter_list>

                 |<name> ( )

<parameter_list> ::= <general_expression> , <parameter_list>

                 | <general_expression>
```

4.2. Examples.

```
sin(5)
addlist(cus,lcus)
uniqlist(lcus)
random( )
```

4.3. Semantics.

Functions designators define single numerical, logical, string, record or structured values which results throught the application of given sets of rules defined in a function, to fixed sets of general expression.

4.3.1. Standard functions.

Certains names should be reserved for the standard functions of analysis. It is recommended that this reserved list should contain : abs, sqrt, sin, cos, int, ...

4.3.2. Transfer functions.

It is understood that transfer functions between any pair of quantities and expressions may be defined. Among the standard functions it is recommended there be one, namely int(k), which transfers an expression of real type to one of type integer.

4.3.3. List manipulation.

In order to manipulate list variables we need these functions :

- addlist : which allows one to add an element or a list of element to a list variable.
- sublist : which allows one to remove an element or a list of element, of a list variable.
- uniqlist : which allows one to reorganize the list so as to eliminate the duplicates.

5. GENERAL EXPRESSION.

5.1. COLLECTION EXPRESSION.

5.1.1. Syntax.

```
<collection_expression> ::= <range>
                          | <list>
                          | <DB_object_set>

<range> ::=                 [ <general_expression> : <general_expression> ]
                          |  [ * : <general_expression> ]
                          |  [ <general_expression> : * ]
                          |  [*:*]

<list> ::=                  { <list_enumeration> }
<list_enumeration> ::=      <list_element> , <list_enumeration>
                          | <list_element>
<list_element>  ::=         <general_expression>
```

```
<DB_object_set> ::=       <variable>  <predicate>

<predicate> ::= ( <coll_cond_factor> ) OR ( <predicate> )

              | <coll_cond_factor>

<coll_cond_factor> ::= ( <coll_cond_term> ) AND ( <coll_cond_factor> )

              | <coll_cond_term>

<coll_cond_term> ::= NOT <coll_cond_primary>

              | <coll_cond_primary>

<coll_cond_primary> ::= <relation_condition>

              | <belonging_cond>

              | ( <predicate> )


<relation_condition> ::=  <relation_operator> <general_expression>

<relation_operator> ::=  <name> : <co>

                         | : <co>


<belonging_cond> ::=     <bel_op>  <general_expression>

<bel_op> ::=             <test_operator>

                         | IN   | NOT IN

<co> ::=                 <cardinal> | <ordinal> | <empty>

<cardinal> ::=           <range> | *

<ordinal> ::=            # <general_expression>

                         | #*
```

5.1.2. Semantics.

In the following sections, we shall call a simple value expression, a general expression whose evaluation -during run time, gives only one value.

5.1.2.1. Range.

A range denotes the set of ordered values between the maximum value (right part), and the minimum value (left part). These values are called the boundaries. Each boundarie must be denoted by simple value expression whose type is integer. '*' used as maximum value means infinity, while '*' used as minimum value means minus infinity.

5.1.2.2. List.

A list consists of an ordered set of simple value expressions. Each simple value expression denotes one element of the list. Two consecutive expression a separated by comma. The overall elements are enclosed between '{}'.

5.1.2.3. Data Base object sets.

Summary.

Generally speaking, a DB object set is defined by a variable followed by a predicate. The variable must denotes a predifined set of DB objects. The predicate acts as a filter on the DB objects of the set denoted by the variable. Therefore, the new defined DB object set is a subset of the predefined DB object set denoted by the variable.

The goal of the DB object set, is, thus, to allow one to define a particular DB object set, from a predifined DB obect set, by means of a predicate.

The first section will present the variables that denotes predefined object sets.

The second section will present the different forms of predicates.

5.1.2.3.1. Variables that denotes predefined DB object sets.

Terminology.

First, we need to define what is a DB object. A DB object is either a record, or a data item value. A record is an occurence of a record type, a data item value is an occurence of a data item.(a detailed description of the DB objects was given in the previous chapter).

Since we consider two type of DB objects, we will have to deal with two types of DB object sets : the RECORD SETS, and the DATA ITEM VALUE SETS.

Variables that denotes predefined RECORD SETS.

The first kind of variable that denotes predifined record set is a fictious variable which is assumed to contain all the records ,of a given DB, that have the same type. The name of the variable is the name of the record type. This kind of variable is implicitly declared when one declared the data base schema used. Thus, for a data base schema that contains five record types, they will be five variables implicitly declared. Each one is assumed to contain all the records of the data base that belong to the record type it denotes. This kind of variable can not be modify by means of assignment statements.

The second kind is an internal variable whose type is

record <name>,

where <name> is the name of the record type that the variable is allowed to contain. This variable is not a very powerful 'set' variable in the sense that it can not contain more than one element.

That's why we need the third kind of variable. It is an internal variable declared as

list of record <name>

where <name> the name of the record type that the elements of the list are allowed to contain.

Example :

CUSTOMER ,which represents all the records whose type are 'customer'.

CUS ,declared as record CUSTOMER, which denotes a variable which is allowed to contain one record whose type is 'customer'.

CUS_SET ,declared as 'list of CUSTOMER', which denotes a list variable whose elements are allowed to contain a record whose type is CUSTOMER.

Variable that denotes predefined DATA ITEM VALUE SETS.

There is only one kind of variable that denotes data item value set. It is a fictious variable implicitly declared with the schema declaration. This fictious variable is assume to contain all the possible values that the data item can take.

5.1.2.3.2. Predicates.

Summary.

The predicate express the particular properties that an element, from the designated predefined DB object set, must verify to be an element of the new defined DB object set. The predicate is an expression that may contain several conditions. These conditions are called collection condition primary. One may combine the collection condition primaries with the binary operators 'OR', and 'AND'. The unary operator 'NOT' is allowed too.

A collection condition primary that checks the value of an object is called a BELONGING CONDITION, while a collection condition primary that checks the relationships of an object with espect to other objects is called a RELATIONSHIP CONDITION.

For convenience, we will call 'ORIGINAL SET', the predefined DB object set denoted by a variable.

1) The belonging condition.

Since there are two kinds of objects, they will be two kinds of belonging conditions.

a) The belonging condition on data item.

The check is made on the value of a data item value with respect to the value of the general expression.

If the belonging operator is a test operator, the general expression must be a simple value expression. Then, a data item value (from the original set) is selected if it verifies the test operator with respect to the value of the general expression.

If the belonging operator is 'IN', or 'NOT IN', the general expression is considered as a set of values. Then, a data item value is selected if it belongs ('IN'), or if it does not belong ('NOT IN') to the set of values denoted by the general expression.

Examples :

1) TOWN = 'ANN ARBOR' defines the set of values (from the data item value set variable TOWN) which is the singleton {'ANN ARBOR'}.
The type of the expression is data item value set.
In this example the type of the elements is string.

2) QUANTITY < 10 defines the set of values (from the data item value set variable QUANTITY)which is {0, 1, ..., 10}.
The type of the expression is data item value set.
In this example the type of the elements is integer.

Remark :

Since there is no internal variable that are allowed to contain data item value set, the data item value set is used as intermediate value in bigger expression, or in iterative statement. It can be stored only if the set defined is a singleton. In that case the determination of the type of such an expression is done with respect to the type of the unique element.

b) The belonging condition on records.

The only operators allowed are 'IN', and 'NOT IN'. The general expression must denote a record set. Then, a record of the original set will be selected if it belongs ('IN'), or if it does not belong ('NOT IN') to the record set denoted by the general expression.

2) The relationship condition.

terminology.

The objects of the data base are related to each other. A record 'customer' may be related to several records 'order'. A data item value 'quantity' may be related to a record 'product', etc ... Therefore, a particular object of the database may be related -throught a particular relationship, to a set of objects. This set is called the **SET OF TARGETS**. A set of targets is indentified by an object and a relationship. The sets of targets are sorted. This means that each element of a set of targets has a rank.

The relationship condition uses the concept of set of targets. Besides, it uses the concepts of original set, and what we call reference set, which is denoted by the general expression. This reference set must contain the same type of objects as the set of targets.

1. If no <co> constraint is specified, the db object set using a relationship condition, defines a set of elements E1, E2, ... ,En that have the following properties :

   1) $\forall i \in \{1, 2, ..., n\}$, Ei belongs to the original set.

   2) Assuming that { Ti1, Ti2 , ... , Tim} is the ordered set of targets of Ei, there is at least a 'j' $\in \{1, 2, ..., m\}$ so that Tij belongs to the reference set.

2. The cardinal constraint is represented by a range. The db object set using a relationship condition that contains a cardinal constraint, defines a set of elements E1, E2, ..., En that have the following properties :

   1) $\forall i \in \{1, 2, ..., n\}$, Ei belongs to the original set.

   2) Assuming that { Ti1, Ti2 , ... , Tim} is the ordered set of targets of Ei,
   assuming that 'k' is the number of Ti that belong to the reference set, then 'k' belongs to the range denoted by the cardinal.

3. The ordinal is represented by a crossroad followed by a simple value expression. Let's call 'ord' the value of the simple value expression. The db object set using a relationship condition that contains a ordinal constraint, defines a set of elements E1, E2, ..., En that have the following properties :

   1) $\forall i \in \{1, 2, ..., n\}$, Ei belongs to the original set.

   2) Assuming that { Ti1, Ti2, ..., Tim} is the ordered set of targets of Ei, Ti,ord belongs to the reference set.

   remark : if the ordinal is #*, then Ti,ord is Tim (the last one).

a) The original set contains records, the reference setcontains records.

Since the relationship correspond to an access path type, the relation operator look like (<name> : , where <name> is the name of the type of the access path that relates a record from the original set to one or more records of the reference set. A record of the original set is selected if there is an access path between the latter and ´some´ records of the reference set.(the precise meaning of ´some´ depends on the <co> constraint).

Examples :

ORDER (O_L : [3:4] ORDER_LINE) defines the set

of records whose types are ORDER which are linked

via the access path type O_L, to three or four

records whose types are ORDER LINE.

The type of the expression is list of record ORDER.

ORDER (O_L : #* ORDER_LINE(:VAL > 1000)) defines

the set of records whose types are ORDER that are

linked to one or more records whose types are

ORDER LINE, so that the last one contains a

data item value VAL greater than 1000.

The type of the expression is list of record ORDER.

b) The original set contains data item values, the reference set contains records.

The resulting set will be a set of data item values. The data item values selected verify the following properties :

- they belong to the original set.
- they are contained in record(s) that belong(s) to the reference set.
- they verify the <co> constraint.

Example :

NAME (: [1:*] CUSTOMER) defines the set of values (of the data item value set variable NAME) which are contained in one or more record whose types are CUSTOMER. Therefore, it defines the set of names of CUSTOMER.

The type of the expression is data item value set.

In this example the type of the elements is string.

c) The original set contains records, the reference set contains data item value.

The resulting set will be a record set which contains record that verify the following properties :

- they belong to the original set.

- they contain data item value(s) that belong(s) to the reference set.

- they verify the <co> constraint.

Example :

ORDER_LINE(:VAL > 1000)) defines the set of

records whose types are ORDER LINE that contains

a data item value VAL greater than 1000.

d) The original set contains data item values, the reference set contains data item values.

This expresion is used with decomposable data items. The resulting set will be a data item value set. The data item values that belong to the resulting set verify the following properties :

- they belong to the original set.
- they contain data item value(s) that belong(s) to the reference set.

- they verify the <co> constraint.

Example :

adr(:city = 'NY') defines the set of data item values 'adr'

that contain at least one data item 'city' whose value

is 'NY'.

5.2. ARITHMETIC EXPRESSION.

5.2.1. Syntax.

```
<arithmetic_expression> ::= <factor> <adding> <arithmetic_expression>
                          | <adding> <factor>
                          | <factor>
<factor> ::=    <term>  <multiplying>  <factor>
              | <term>
<term> ::=      <primary>  ^  <term>
              | <primary>
<primary> ::=   <string>
              | <unsigned_number>
              | <collection_expression>
              | <func>
              | <variable>
              | ( <arithmetic_expression> )
<unsigned_number> ::= <unsigned_integer>
              |  <unsigned_real>
<adding> ::=    + | -
<multiplying> ::= * | /
```

5.2.2. Examples.

```
'Hello ' + 'the ' + 'world.'
sqrt(sin(a+b))
customer(c_o : [1:*] order)
```

5.2.3. Semantics.

An arithmetic expression is a rule for processing informations. The result is obtained by executing the indicated operations on the actual values of the primary of the expression. The actual value of a primary is obvious in the case of unsigned number, string constant or logical constant (true, false). For variables, it is the current value. For function designator, it is the value arising from the computing rules defining the function when applied to the current values of the procedure parameters given in the expression. For collection designation, it is the set of values wich verify the predicates of the collection designation .

Finally, for arithmetic expressions enclosed in parentheses, the value must -through a recursive analysis, be expressed in terms of the values of the primaries of the other four kinds.

5.2.3.1. Operators and type.

The type of the primary is obvious if it is a string constant, or an unsigned number . For a collection expression, the type is either 'record', 'list of record' or 'data item value set'. For functions, it is the type declared in the declaration part (section 7.1). For variable, one may refers to the declaration part (section 7.2). Simple variables have only one type. The type of structured variables which are taken in a global way, is either LIST, RECORD, ARRAY, or STRUCTURE. On the other hand, if the structured variable indicates one of its elements, the type of the primary is the type of the element.

5.2.3.2. Precedence of operators.

- first : ^
- second : * /
- third : + -

5.3. GENERAL EXPRESSION.

5.3.1. Syntax.

```
<general_expression> ::= <general_factor> OR <general_expression>
                         | <general_factor>
<general_factor> ::= <general_term> AND <general_factor>
                         | <general_term>
<general_term> ::=       NOT <general_primary>
                         | <general_primary>
<general_primary> ::= <logical_value>
                         | <test_expression>
                         | <arithmetic_expression>
                         | <co> <collection_designation>
<test_expression> ::= <arithmetic_expression>  <test_operator>
                              <arithmetic_expression>
```

5.3.2. Examples.

```
a + b
true
a > b
```

5.3.3. Semantics.

The principles of evaluation of general expression are entirely analogous to those given for arithmetic expression in section 5.2.3 .

5.3.3.1. Operators and type.

The general expresion may be just an arithmetic expression. In that case, the type of the general expression is obviously the type of the arithmetic expression. The operators OR, AND, NOT accepts only expressions whose type are BOOLEAN.

5.3.3.2. Precedence of operators.

The sequence of operations within one expression is generally from left to right, with the following additional rules :

- first : collection expression
- second : arithmetic expression.
- third : < =< = >= > =/=
- fourth : NOT
- fifth : AND
- sixth : OR

The use of parentheses will be interpreted in the sense given in section 5.2.3 .

6. STATEMENTS.

6.1. STATEMENT PART, COMPOUND STATEMENT.

6.1.1. Syntax.

```
<statement_part> ::= BEGIN <statements> END

<statements> ::= <statement>  ;  <statements>

               | <statement>

<statement> ::= <ass_st>

              | <for_st>

              | <next_st>

              | <exit_st>

              | <while_st>

              | <db_mod>

              | <if_st>

              | <call_st>

              | <return_st>

              | <dummy>

<dummy> ::=     <empty>
```

6.1.2. Examples.

```
BEGIN
        i := 3;
        p := 'hello the world'
END.
```

6.2. ASSIGNMENT STATEMENT.

6.2.1. Syntax.

<ass_st> ::= <variable> := <general_expression>

6.2.2. Examples.

```
var := date.year

var := date[3]

var := year(date)

var := name(:customer(:numcus = 1345))
```

6.2.3. Semantics.

Assignment statements serve for assigning the value to one variable. The type of the left part must be the same as the type of the right part. Moreover, for structured variable taken in a global way, not only the type, but also the substructure must be the same. A general expression whose type is data item value set can be assigned to a variable only if the set is a singleton. Then, the type checking is done between the type of the left part and the type of the element indicated in the right part.

6.3. FOR STATEMENT.

6.3.1. Syntax.

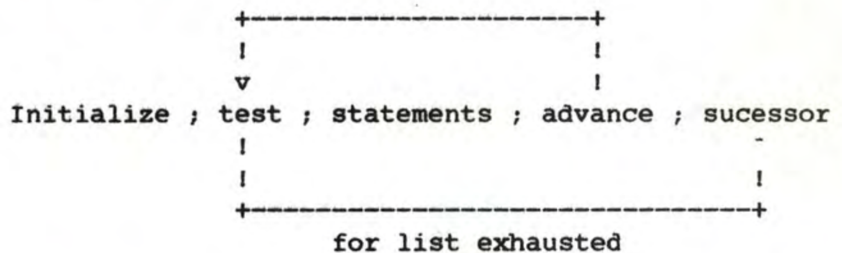
```
<for_st> ::= for <variable> := <for_list> do <statements> endfor

<for_list> ::=  <general_expression> <ord>

               | <general_expression>

<ord> ::=       order <order_keys>

<order_keys> ::= <name>  <ad> , <order_keys >

               | <name> <ad>

<ad> ::=        ascending | descending | <empty>
```

6.3.2. Examples.

```
for I:= [20:30]
do
        print(I)
endfor

for CUS := CUSTOMER  order NAME ascending
do
        print(NAME(:CUS))
endfor
```

6.3.3. Semantics.

The enumerative loop orders the execution of a sequence of statements. In addition, it performs a sequence of assignment to its controlled variable. The process may be visualized by means of the following picture :



In this picture the word initialize means : perform the first assignment of the for clause. Advance means : perform the next assignment of the for clause. Test determines if the last assignment has been done. If so, the execution continues with the successor of the for statement. If not the statement following the for clause is executed.

6.3.3.1. The for list elements.

The for list gives a rule for obtaining the values which are consecutively assigned to the controlled variable. This sequence of values is obtained from the for list elements by taking these only one by one, in the specified order.

The general expression must defined a set of values which have the same type as the controlled variable.

The clause order allows one to specify a particular order in the selection of the for list elements.

6.4. NEXT AND EXIT STATEMENTS.

6.4.1. Syntax.

<next_st> ::= next <name> | next

<exit_st> ::= exit <name> | exit

6.4.2. Semantics.

The EXIT statement allows one to abort the execution of a loop. If no name is given the abortion concerns the inmost loop. If a name is specified, it must be the name of a loop control variable. Then, the abortion concerns the loop controled by the specified variable, and, obviously, all the loops inside the latter.

The NEXT statement causes the termination of the execution of the body of the loop. Then the loop is in the state advance (fig above). If no name is given the termination concerns the inmost loop body. If a name is specified, it must be the name of a loop control variable. Then, the termination concerns the loop controled by the specified variable. This termination implies the abortion of all the loops inside the latter .

6.5. WHILE STATEMENT.

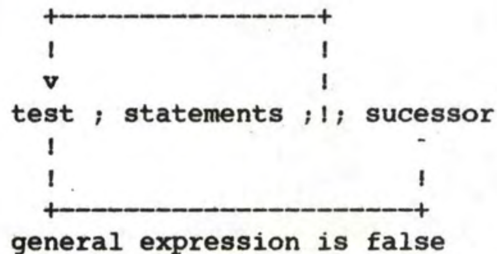
6.5.1. Syntax.

<while_st> ::= while <general_expression> do <statements> endwhile

6.5.2. Examples.

6.5.3. Semantics.

The while loop orders the execution of a sequence of statements as the enumerative loop but it does not perform any assignment. The process may be visalized by the means of the follwing picture :



The while statement does not use any controlled variable. If the evaluation of the general expression gives FALSE, it continues with the successor of the while statement. If the result is TRUE, the statement following the while statement will be executed.

Obviously, the type of the general expression must be boolean.

6.6. IF STATEMENT.

6.6.1. Syntax.

```
<if_st> ::=     if <general_expression> then <statements>  endif

              | if <general_expression> then <statements>
                else <statements> endif
```

6.6.2. Examples.

```
1) if [2..*] CUSTOMER( :NAME = 'SMITH' )
   then  print('Duplicates smith' )
   endif

2) if A > B or B > C
   then  print('yes' )
   else  print('no' )
   endif

3) if I = J
   then         if J = K
                then   print('i=j and j=k' )
                else   print('i=j and j not = k' )
                endif
   endif

4) if I = J
   then         if J = K
                then   print('i=j and j=k' )
                endif
   else         print('i not = j' )
   endif
```

6.6.3. Semantics.

Conditional statements cause certain statements to be executed or skipped, depending on the running values of the general expression. Obviously, the type of these general expression must be boolean.

The statements following the THEN clause will be executed if the evaluation of the general expression gives TRUE, otherwise these statements will be skipped, and the statements following the ELSE clause will be executed, if any.

Because of the ENDIF clause, there will be no ambiguity if there are other IF statements inside the main one.

We assume that a ENDIF clause will always refer to the last encounted IF clause.

6.7. CALL AND RETURN STATEMENTS.

6.7.1. Syntax.

```
call_st ::=     <name> ( )

                | <name> ( <parameter_list> )

return_st ::=   return

                | return ( <parameter_list> )
```

6.7.2. Semantics.

The call statement allows one to invoke a procedure or a function. The procedures invoked must be declared in the procedure declaration. The functions invoked must be declared in the function declaration.

In an invoked procedure (or function), the return statement order the come back to the main program.

6.8. DATA BASE MODIFICATION STATEMENTS.

These statements work on the data base whose schema is defined in the extern declaration part .

```
<db_mod> ::=    <modif>
              | <creat>
              | <del>
              | <att>
              | <det>
              | <transf>
```

6.8.1. MODIFY STATEMENT.

6.8.1.1. Syntax.

```
<modif> ::=    modify <variable> <relation_cond>
```

6.8.1.2. Semantics.

The variable must denotes a record set. The relationship condition must be a relationship condition on data items. Therefore, the item values of the records contained in the variable will be modified in such a way that they will verify the relationship condition.

6.8.2. CREATE STATEMENT.

6.8.2.1. Syntax.

```
<creat> ::=    create <variable> := <variable> <create_cond>

<create_cond> ::= <belonging_cond>
                | <relation_cond>
```

6.8.3. Semantics.

The variables must denote a record set. Therefore, a record is created in such a way that it will verify the create condition. This record is assigned to the left variable.

6.8.4. DELETE STATEMENT.

6.8.4.1. Syntax.

<del> ::= delete <variable>

6.8.4.2. Semantics.

The variable must denotes a record set. Therefore, the indicated set of records are deleted. The records linked by an existence constraint to the deleted records, are also deleted, and so on.

6.8.5. ATTACH STATEMENT.

6.8.5.1. Syntax.

<att> ::= attach <variable> to <variable> via <name>

6.8.5.2. Semantics.

The variables must denotes a record set. The <name> must refers to an access path type. Therefore, the set of records indicated by the left variable will be linked via the specified access path type, to the set of records indicated by the right variable -if the linkage is allowed in the running schema. If there is an inverse path type, it is updated.

6.8.6. DETACH STATEMENT.

6.8.6.1. Syntax.

<det> ::= detach <variable> from <variable> via <name>

| detach <variable> via <name>

6.8.6.2. Semantics.

The type of the variable must denotes a record set. The <name> must refers to an access path type. Therefore, the specified access path types between the set of records indicated by the left variable and the set of records indicated by the right variable, will be deleted. If there is an inverse path type, it is updated.

6.8.7. TRANSFER STATEMENT.

6.8.7.1. Syntax.

```
                                (1)              (2)
<transf> ::=     transfer <variable> from <variable>
                       (3)
                 to <variable> via <name>

                 | transfer <variable> to <variable> via <name>
```

6.8.7.2. Semantics.

The type of the variables must denotes a record set. Therefore, the transfer statement is similar to :

```
               (1)              (2)
   detach <variable> from <variable> via <name>
               (1)              (3)
   attach <variable> to   <variable> via <name>
```

Since TRANSFER is an atomic action, it is useful while manipulate record on which there is a existence constraint, because, in that case one is not allowed to use a DETACH followed by an ATTACH.

7. DECLARATIONS.

Declarations serve to define certains properties of the informations used in the program, and to associate them with identifiers (<name>).

<declarations> ::= <extern_definition_part>

<variable_definition_part>

7.1. EXTERN DEFINITION PART.

7.1.1. Syntax.

<extern_definition_part> ::= <schema_declaration>

<algorithm_declaration>

<function_declaration>

<schema_declaration> ::= schema <name> | <empty>

<procedure_declaration> ::= procs <names>

| <empty>

<function_declaration> ::= funcs <name_list>

| <empty>

<name_list> ::= <name> : <variable_type> , <name_list>

| <name> : <variable_type>

7.1.2. Examples.

```
schema  ACCOUNT-SCHEMA
procs   VERIF_COUNT, ERROR_COUNT
funcs   sin : real, sqrt : real
```

7.1.3. Semantics.

Schema declaration.

A schema is a description of a database structure. This description is stored outside the algorithms. The schema declaration indicates the name of the particular description to which the algorithm will refer. Besides, it declares (implicitly) all the fictuous variables associated with the schema (section 5.1.2.3.1).

Procedure declaration.

Procedure are ADL algorithms that are used by other algorithms. The procedure declaration serve to indicate which procedures are called in the algorithm.

Function declaration.

The function declaration indicates the names of the functions used, and the type of the returned values.

7.2. VARIABLE DECLARATION PART.

7.2.1. Syntax.

```
<variable_definition_part> ::= <empty>

                             | var <variable_declarations>

<variable_declarations> ::= <variable_declaration>

                             | <variable_declaration><variable_declarations>

<variable_declaration> ::= <names> : <variable_type>

<names>   ::=     <name_lst>

<name_lst> ::=  <name> , <name_lst>

                | <name>

<variable_type> ::= <simple>

                | <structured>

<simple> ::=    boolean

                | string

                | real

                | integer

<structured> ::= <structure >

                | <array>

                | <db_record>

                | <list>

<structure> ::= structure <field_list> end

<field_list> ::= <variable_declaration> <field_list>

                | <variable_declaration>

<array> ::=     array [ <index_list> ] of <component_type>

<index_lst> ::= <index> , <index_lst>

                | <index>

<index> ::=     <unsigned_integer>

<component_type> :: <simple>
```

```
                | <structure>

                | <db_record>

                | <list_of>

<db_record> ::=  record <name>

<list_of> ::=    list of <variable_type>
```

7.2.2. Examples.

```
   var     I,J :    integer

           TAB :    array [3,6,4] of record PRODUCT

           CUS :    list of  CUSTOMER

           REC :    structure
                            FIELD1 : integer
                            FIELD2 : real
                            FIELD3,FIELD4 : boolean
                    end
```

7.2.3. Semantics.

A variable declaration declares one or several identifiers (<name>) to represent simple or structured variables.

7.2.3.1. Simple variables.

Simple variables contain only one information. Real declared variables may only assume positive or negative values, including zero. Integer declared variables may only assume positive and negative integral values, including zero. Boolean declared variables may only assume the values TRUE and FALSE. The string declared variables may only assume sequence of characters.

7.2.3.2. Structured variables.

7.2.3.2.1. Structure variable.

Structure and record declared variables may contain several informations at the same time. Each information is stored in a separate field. One can access the field through its name. The field decompostion declaration of a structure variable is done in the declaration part of the algorithm.

7.2.3.2.2. Arrays.

An arrays declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables, and gives the dimensions of the arrays, the bounds of the subscripts and the types of the variables.

7.2.3.2.3. Record variables.

The record variables serve to keep occurences of database records. Thus the <name> specified must be a record type that belongs to the schema declared previously. It indicates the type of the record that the variable is allowed to contain.

7.2.3.2.4. Lists.

Lists are formed from elements. Each element of the list may be considered as fictuous variable. All the elements must have the same type.

Index.

ad, 29
adding, 23
arithmetic_expression, 23
arithmetic_operator, 6
array, 39
ass_st, 28
att, 35
basic_symbol, 5
bel_op, 13
belonging_cond, 13
bracket, 6
call_st, 33
cardinal, 13
co, 13
coll_cond_factor, 13
coll_cond_primary, 13
coll_cond_term, 13
collection_expression, 12
component_type, 39
creat, 34
create_cond, 34
db_mod, 34
db_object_set, 13
db_record, 40
declarations, 37
declarator, 6
del, 35
delimiter, 6
det, 35
digit, 5
dummy, 27
exit_st, 30
exponent_part, 7
extern_definition_part, 37
factor, 23
field_list, 39
field_selection, 9
for_list, 29
for_st, 29
func, 11
function_declaration, 37
general_expression, 25
general_factor, 25
general_primary, 25
general_term, 25
hierac_variable, 9
if_st, 32
index, 39
index_lst, 39
letter, 5
list, 12
list_element, 12
list_enumeration, 12
list_of, 40
logical_operator, 6
logical_value, 5
modif, 34
multiplying, 23
name, 7
name_list, 37
name_lst, 39
names, 39
next_st, 30
non_hierac_variable, 9
operator, 6
ord, 29
order_keys, 29
ordinal, 13
parameter_list, 11
pas_declaration, 37
predicate, 13
primary, 23
procedure_declaration, 37
range, 12
real_number, 7
relation_condition, 13
relation_operator, 13
relationship_operator, 6
return_st, 33
separator, 6
sequential_operator, 6
set_belonging_operator, 6
sign, 7
simple, 39
statement, 27
statement_part, 27
statements, 27
string, 8
structure, 39
structured, 39
subscript_list, 9
subscripted_variable, 9
term, 23
test_expression, 25
test_operator, 6
transf, 36
unsigned_integer, 7
unsigned_number, 23
unsigned_real, 7
variable, 9
variable_declaration, 39
variable_declarations, 39
variable_definition_part, 39
variable_type, 39
while_st, 31

Chapter 4: STATISTICAL MODEL

1. Introduction

The goal of the statistical model is to provide the database designer with a set of concepts which enable him to build the statistical description of a GAM database schema (see chapter 2).

In this chapter, we will first present the statistical model choosed by Anne KERSTENNE (see [KER.83]) and later we will give the E-R schema of the model.

2. Presentation of the statistical model

The model is based on the statistical description of the relation between two object types : R(A,B). We distinguish between two kinds of object types :

- Simple object type : non decomposable data items.
- Complex object type : record types or decomposable data items.

We consider only relations between two complex object types or between one complex object type and one simple object type. A relation between two complex object types corresponds to an access path type or a decomposable item. A relation between one complex object type and one simple object type corresponds to a non decomposable item (see KERSTENNE 83).

Notations :

- Let NA be the number of occurrences of the object type A in the database.
- Let NB be the number of occurrences of the object type B in the database.
- Let ID be the minimum number of occurrences of the object type B related by R to one occurrence of the object type A.
- Let JD be the maximum number of occurrences of the object type B related by R to one occurrence of the object type A.
- Let II be the minimum number of occurrences of the object type A related by R to one occurrence of the object type B.
- Let JI be the maximum number of occurrences of the object type A related by R to one occurrence of the object type B.

NA, NB, ID, JD, II, JI are associated to each relation R(A,B).

In order to describe the R(A,B) relation, we choosed to use two distribution functions :

- The direct function :

  FD(k)   ID <= k <= JD , k integer

  0 <= FD(k) <= 1

  FD(k) is a real function giving the proportion of occurrences of object type A related by R to exactly k occurrences of object type B.

- The inverse function :

  FI(k)   II <= k <= JI , k integer

  0 <= FI(k) <= 1

  FI(k) is a real function giving the proportion of occurrences of object type B related by R to exactly k occurrences of object type A.

The description of the relation between two complex object types consists of :

NA, NB, ID, JD, II, JI, FD(k), FI(k).

In order to describe the distribution of data item values, we need a proportion function :

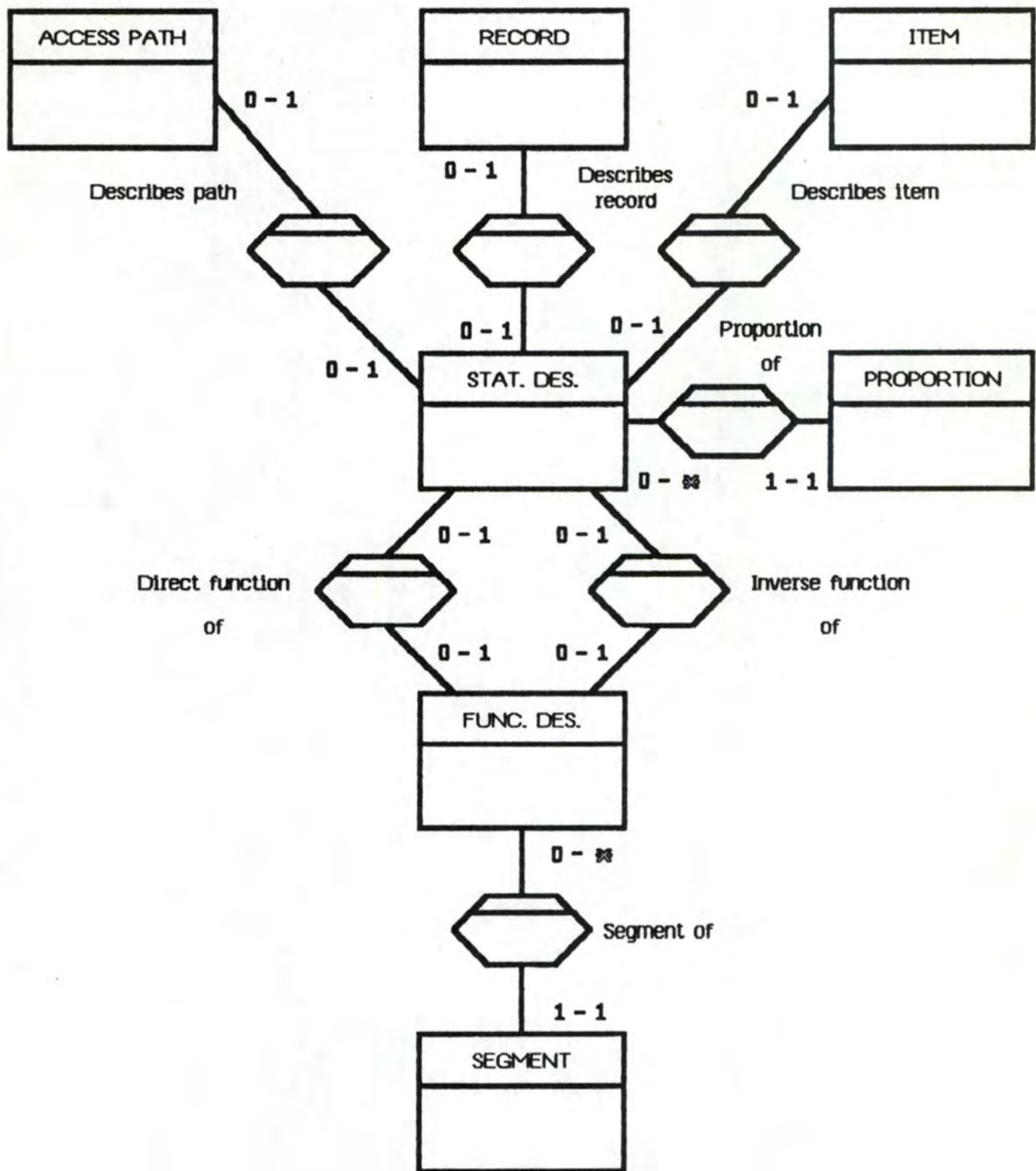
- Let R(A,B) be a relation between one complex object type A and one data item B.

- Let b be a possible value of the data item B.

- PR(b) gives the proportion of occurrences of object type A related to one occurrence of object type B having the b value. PR(b) is merely the proportion of occurrences of object type A which have their B data item equal to b.

Depending on the type of value (real, integer, alphanumeric, etc.), PR(b) may be given for range of values.

3. E/R schema of the statistical model.



We use four entities and seven relationships to collect statistical information :

Entities

- STATISTICAL DESCRIPTION
- FUNCTION DESCRIPTION
- SEGMENT
- PROPORTION

Relationships

- DESCRIBES ITEM
- DESCRIBES PATH
- DESCRIBES RECORD
- PROPORTION OF
- DIRECT FUNCTION OF
- INVERSE FUNCTION OF
- SEGMENT OF

We will give a description of each of these concepts and the integrity constraints.

3.1. ENTITIES

3.1.1. STATISTICAL DESCRIPTION

This entity describes the statistical part of an access path type, or a data item, or a record type. It used as an interface between the described GAM object types and the three others entities which compose the statistical information.

Properties :

- TYPE-P : if the entity describes the relation between one complex object type and one simple object type, TYPE-P gives the type of the data item values (real, integer, alphanumeric, alphanumeric with limited set of values).
- NUM-ELEMENTS : if the entity describes a record type, NUM-ELEMENTS is the number of occurrences of the record type in the database.

3.1.2. FUNCTION DESCRIPTION

This entity represents the description of one distribution function. It collects the statistical information about the direct or (excluding) inverse distribution function of a relation R(A,B).

An actual distribution function will be chosen in a set of 14 predefined distribution functions. The functions of the set have been carefully selected to allow the approximation of almost any real function. In the following sections, we will present the predefined functions. We will use F(k) to denote a distribution function (FD(k) or FI(k)), I to denote ID or II, J to denote JD or JI (see section 2). The letter m will be used to denote the average number of occurrences of object type B related to one occurrence of object type A.

Function 1 : Real function

Used if F(k) is known.

[I,J] is divided into n separate classes [I(t),J(t)].
$1 <= t <= n$
$I(1) = I \quad J(n) = J$
$I(i) = J(i-1) + 1$

F1(t) = proportion of occurrences of object type A related to k occurrences of object type B with $I(t) <= k <= J(t)$.

Function 2 : Binomial function

If m and J are known :

Let $p = m/J$

$$F2(k) = \binom{J}{k} p^{k} (1-p)^{J-k} \qquad 0 <= k <= J$$

Function 3 : Geometric 0

If m value is known :

Let $p = 1/(m+1)$

$$F3(k) = p(1-p)^{k} \qquad k >= 0$$

Function 4 : Geometric 1

If m value is known and higher than 1 :

Let $p = 1/m$

$$F4(k) = p(1-p)^{(k-1)} \qquad k >= 1$$

<u>Function 5</u> : <u>Poisson function</u>

If m value is knowm :

$$F5(k) = \exp(-m) \frac{m^k}{k!} \qquad k >= 1$$

<u>Function 6</u> : <u>Uniform 1</u>

If I and J values are known :

Let p = 1/(j-i+1)

F6(k) = p   I <= k <= J

<u>Function 7</u> : <u>Uniform 2</u>

If J and F(0) are known :

Let p = (1-F(0))/J

F8(k) = p   1<= k <= J

<u>Function 8</u> : <u>Uniform 3</u>

If m value is known and I = 0 :

Let p = 1/(2m+1) , J = 2m

F8(k) = p   0 <= k <= J

<u>Function 9</u> : <u>Uniform 4</u>

If m value is known and I = 1 :

Let p = 1/(2m-1)   J= 2m-1

F9(k) = p   1<= k <= J

<u>Function 10</u> : <u>Decreasing function</u>

If m, J and F(0) are known :

$$F10(k) = a \exp(-bk) \quad 1 <= k <= J$$

a and b are given by two equations (see [KER.83] page 16-17)

<u>Function 11</u> : <u>Maximum 1 function</u>

Let k0 be the value of k where F(k) is maximum.

If m, J, F(0) and k0 are known :

$$F11(k) = a \exp(-b \, abs(k-k0)) \quad 1 <= k <= J$$

a,b are given by the same equations as above (function 10)

<u>Function 12</u> : <u>Maximum 2 function</u>

If m, J, F(0), k0 (see function 11) are known :

$$F12(k) = a(1-\exp(-bk)) \quad 1 <= k <= k0$$

$$F12(k) = a(1-\exp(-bk0)) \exp(-b(k-k0)) \quad k0 <= k <= J$$

See [KER.83] (page 17, 18) for a, b values

<u>Function 13</u> : <u>J=1 function</u>

If m is known :

$$F13(0) = 1 - m$$
$$F13(1) = m$$

If F(0) is known :

F13(0) = F(0)
F13(1) = 1 - F(0)

Function 14 : J=1 function

If m and F(0) are known :

F14(0) = F(0)
F14(1) = 2(1-F(0))-m
F14(2) = m-(1-F(0))

Properties :

- FUNCTION-TYPE : gives the type of the function. It identifies one of the above described functions.
- MEAN : gives the mean value of the distribution function. It corresponds to m in the above description of functions.
- F-ZERO : is the value of the distribution function in 0. It corresponds to F(0) in the above description of functions.
- K-MAX : gives the point where the function is maximum. It corresponds to k0 in the above description of functions.
- PAR-ONE : parameter needed for certain of the predifined distribution functions (see the following "Parameter Usage Matrix").
- PAR-TWO : same as PAR-ONE.
- MIN-RELATIONS : gives the value of ID (or II if inverse distribution function).
- NUMBER-TARGET : if the described relation is R(A,B), It gives the number of A (B) objects if the FUNCTION DESCRIPTION entity describes the inverse (direct) distribution function.

PARAMETER USAGE MATRIX

	FUNCTION-TYPE	MEAN	F-ZERO	K-MAX	PAR-ONE	PAR-TWO	MIN-RELATIONS	MAX-RELATIONS	NUMBER-TARGET
REAL-FUNCTION (2)	*								*
UNIFORM-1	*						*	*	*
UNIFORM-2	*		*					*	*
UNIFORM-3	*	*					*		*
UNIFORM-4	*	*					*		*
POISSON	*	*							*
BINOMIAL	*	*						*	*
GEOMETRIC-0	*	*							*
GEOMETRIC-1	*	*							*
DECREASING	*	*	*		*	*		*	*
MAX1-FUNCTION	*	*	*	*	*	*		*	*
MAX2-FUNCTION	*	*	*	*	*	*		*	*
J1-FUNCTION	*	*(1)	*(1)					*	*
J2-FUNCTION	*	*	*					*	*

(1) For J1-FUNCTION, we don´t need both parameters, only one of then is enough.

(2) The REAL FUNCTION is the only one which uses SEGMENTS (see section 3.1.3)

3.1.3. SEGMENT

When the disribution function is a REAL-FUNCTION, we need a variable number of classes to describe this function. We use the SEGMENT entity to describe one class.

Properties :

- MAX-CLASS-S : gives the upper limit of the class.
- VALUE-S : gives the proportion of occurrences of object type A (B if inverse) related to n occurrences of object type B (A if inverse) with n included in the class.

3.1.4. PROPORTION

This entity describes one class of the proportion function PR(b) (see section 2). A proportion function is needed only when one wants to describe a relation involving a non decomposable data item.

Properties :

- MAX-CLASS-P : gives the upper limit of the class.
- VALUE-P : is the proportion of occurrences of object type A which have a B value included in the class.

3.2. RELATIONSHIPS

3.2.1. DESCRIBES ITEM

This relationship denotes the fact that one data item is described by one STATISTICAL DESCRIPTION entity. The connectivity of the relationship is 0-1, 0-1.

3.2.2. DESCRIBES PATH

This relationship denotes the fact that one access path type is described by one STATISTICAL DESCRIPTION entity. The connectivity of the relationship is 0-1, 0-1.

3.2.3. DESCRIBES RECORD

This relationship denotes the fact that one record type is described by one STATISTICAL DESCRIPTION entity. The connectivity of the relationship is 0-1, 0-1.

3.2.4. PROPORTION OF

This relationship denotes the fact that one class of the PR(b) proportion function (see section 2) is described by one PROPORTION entity. The connectivity of the relationship is 1-1 for PROPORTION and 0-* for STATISTICAL DESCRIPTION.

3.2.5. DIRECT FUNCTION OF

This relationship denotes the fact that one FUNCTION DESCRIPTION entity describes the direct distribution function FD(k) (see section 2). The connectivity of the relationship is 0-1 for FUNCTION DESCRIPTION and 0-1 for STATISTICAL DESCRIPTION.

3.2.6. INVERSE FUNCTION OF

This relationship denotes the fact that one FUNCTION DESCRIPTION entity describes the inverse distribution function FI(k) (see section 2). The connectivity of the relationship is 0-1 for FUNCTION DESCRIPTION and 0-1 for STATISTICAL DESCRIPTION.

3.2.7. SEGMENT OF

This relationship denotes the fact that one class of a distribution function of type REAL-FUNCTION is described by one SEGMENT entity.

3.3. INTEGRITY CONSTRAINTS

- A STATISTIC DESCRIPTION entity must be related to one and only one of the following entities :
  - ACCESS PATH by the DESCRIBES PATH relationship
  - RECORD by the DESCRIBES RECORD relationship
  - ITEM by the DESCRIBES ITEM relationship
- If a STATISTICAL DESCRIPTION entity is related to an ACCESS PATH or ITEM entity, it must be related to a FUNCTION DESCRIPTION entity by the DIRECT FUNCTION OF relationship or (excluding) by the INVERSE FUNCTION OF relationship.
- A FUNCTION DESCRIPTION entity must be related to a STATISTICAL DESCRIPTION entity by the DIRECT FUNCTION OF or (excluding) INVERSE FUNCTION OF relationship.
- If the FUNCTION-TYPE property of a STATISTICAL DESCRIPTION entity has the value REAL-FUNCTION, the entity must be related to at least two SEGMENT entities.
- If a STATISTICAL DESCRIPTION entity is related to a non decomposable item, it must be related to at least two PROPORTION entities.

Chapter 5: GENERAL ARCHITECTURE OF A LOGICAL DB DESIGN SYSTEM.

1. Introduction

Logical database design is a step of most currently known database design methodologies. It offers a way to progress in the database design, while staying independent of the target hardware and software. Therefore, a logical database design system should not depend of a particular "target" DBMS.

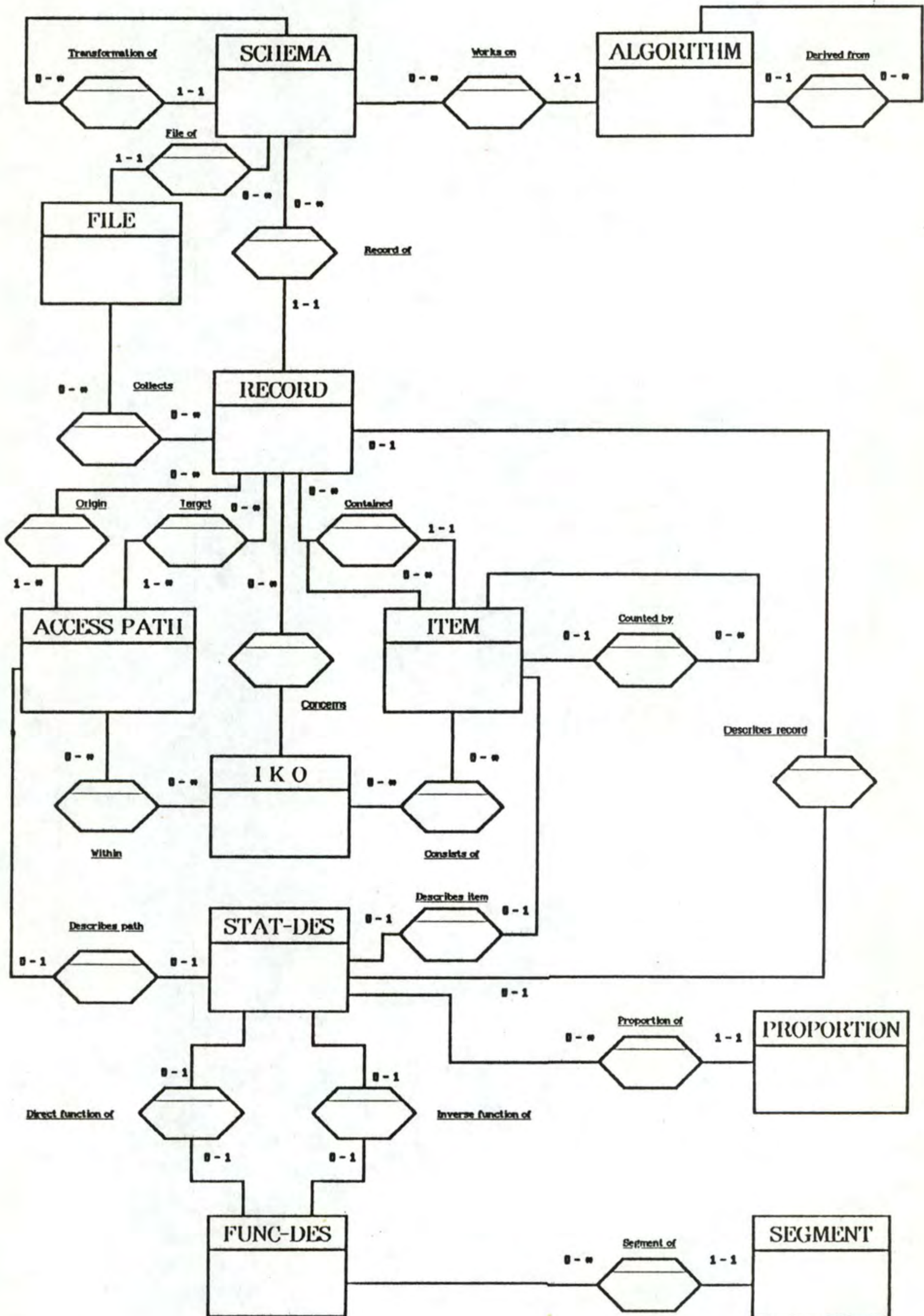
Because logical design is part of a general design methodology (e.g. requirements formulation and analysis, logical design, physical design), a logical database design system should be integrated with design systems supporting conceptual analysis and physical design.

In this chapter, we will :

- present the conceptual schema of the database supporting the logical database design system.

- define the concepts and steps proper to the methodology of Namur.

2. CONCEPTUAL SCHEMA OF THE DATABASE



The conceptual schema is divided into three parts :

- the first part concerns database schemas. It allows the representation of database schemas expressed in the GAM model (see chapter 2). It is composed of the entity SCHEMA and the GAM entities FILE, RECORD TYPE, ACCESS PATH TYPE, and DATA ITEM. As it is explained later, the entity IKO represents the GAM concepts of identifier, access key, and order.
- the second part concerns the statistical description of a database schema. The description is based on the statistical model (see chapter 4). It consists of the entities STATISTICAL DESCRIPTION, FUNCTION DESCRIPTION, PROPORTION, SEGMENT.
- The third part allows the representation of access algorithms. It is composed of the entity ALGORITHM.

We will not describe the statistical part of the conceptual schema. This work has been done in chapter 4. For the same reason, we will refer to chapter 2 for the definition of the GAM entities.

2.1. Entities

2.1.1. SCHEMA

It represents a database schema expressed in the Generalised Access Model. A schema is a logical representation of data structures.

Properties

- SCHEMA-TYPE : this property indicates the type of the schema.

2.1.2. ALGORITHM

It represents an access algorithm in ADL language. An algorithm works on the data structure specified by the schema related to it.

Properties

- ALGORITHM-TYPE : this property indicates the type of the algorithm.
- ALGORITHM-TEXT : contains the text of the algorithm in ADL.

2.1.3. FILE

It represents a GAM object FILE (see definition in chapter 2, section 2.4).

2.1.4. RECORD

It represents a GAM object RECORD TYPE (see definition in chapter 2, section 2.1).

2.1.5. ACCESS PATH

It represents a GAM object ACCESS PATH TYPE (see definition in chapter 2, section 2.3)

Properties

- FUNCTIONAL CLASS : It indicates the functional class of the access path type. This property takes four values : one-to-one, one-to-many, many-to-one, many-to-many (see chapter 2 section 4.1).

2.1.6. ITEM

It represents a GAM object DATA ITEM (see definition in chapter 2, section 2.2).

Properties

- OPTIONAL : The value of this property is YES if an occurrence of the associated record type or decomposable data item does not have to be always associated to a value of the data item. The value of the property is NO if the association between record type or decomposable data item occurrences and values of the data item is mandatory..S1
- REPETITIVITY : It indicates if the data item is repetitive or not. A data item is repetitive if more than one value of it can be associated to one occurrence of the associated record type or decomposable data item. The values taken by the repetitivity property are YES or NO.

- FORMAT : description of the data item format in COBOL way.
  FORMAT is a text constant.

  examples :

  999 : integer numbers from 0 to 999.
  X(5) : five alphanumeric characters.
  A(20) : 20 alphabetic characters.

2.1.7. IKO

The IKO is the generalisation of three concepts : Identifier, access Key, Order. These concepts have been presented in the description of the GAM. Identifier is an integrity constraint (see 2 section 4.2). Access key and order are GAM objects (see chapter 2 respectively sections 2.6 and 2.7).

An occurrence of the object type IKO may express one, two or all these concepts at the same time.

Properties

- GLOBAL : Thru the concept of global, we want to extend the domain of validity of an identifier, an acces key , an order to many record types. There are data items in different record types, which contains the same kind of information (for example WORKER-NUMBER and EMPLOYEE-NUMBER are both number of persons). If there is no intersection the set of possible values (WORKER- NUMBER takes values from 1 to 1000, and EMPLOYEE-NUMBER takes values from 1001 to 2000), we can say that both numbers are identifiers for both record types. In order to express this fact, we will say that both IKO belongs to the same global.

  A global is identified by an integer number called "global number". The value of the GLOBAL property is the number of the global to which the IKO belongs. If the IKO does not belong to a global, the value of the property is zero.

- REFERENCE-SET : it indicates the referential type of the IKO (see 3.1.1.6). We consider only three types of referential :
  - DB = all the database schema.
  - EF = each file taken separatly.
  - AP = targets of access paths.

- IDENT-KEY-ORDER : the value of this property is a string composed by 1 character minimum and 3 characters maximum. The allowed alphabet is ("I","K","O"). Although, the characters are facultative, they appear in the order "I","K","O". An "I", "K" or "O" is present depending upon whether the IKO represents respectively an identifier, an order, or a key.

- DUPLICATES : This property indicates if an acces key allows duplicates. The values of the property specify the order of storage of the duplicates elements :

  NO : no duplicates.
  LIFO : duplicates stored in LIFO order.
  FIFO : duplicates stored in FIFO order.

RANDOM : duplicates stored in random order.

- ORDER-TYPE : this text constant describes, in free text format, the type of order.

2.2. Relationships

2.2.1. FILE_OF

This one-to-many relationship from the entity SCHEMA to the entity FILE is used to denote the fact that one file belongs to one schema. The relationship is mandatory for the FILE entity.

2.2.2. RECORD_OF

This one-to-many relationship from the entity SCHEMA to the entity RECORD is used to relate a schema to the record types belonging to it. The relationship is mandatory for the RECORD entity.

2.2.3. COLLECTS

This many-to-many relationship from the entity RECORD to the entity FILE denotes the fact that a record type is collected by a file.

2.2.4. ORIGIN

This many-to-many relationship relates a access path type with its origin record types. The relationship is mandatory for the ACCESS PATH entity.

2.2.5. TARGET

This many-to-many relationship relates an access path type with its target record types. The relationship is mandatory for the ACCESS PATH entity.

2.2.6. CONTAINED

This relationship relates a data item with the record type or decomposable data item to which it belongs. It is a binary relationship. A data item belongs to one record type or (exclusive) to one decomposable data item. The relationship is one to many from the containing entity to the contained entity. The relationship is mandatory for the data item.

2.2.7. COUNTED BY

This one-to-many relationship from the entity ITEM to the entity ITEM is used to denote the fact that a data item is the repetitivity counter of a group of data items.

2.2.8. CONCERNS

This many-to-one relationship from the entity IKO to the entity RECORD relates an IKO to the record type concerned by the identifier, key or order. The relationship is mandatory for the IKO.

2.2.9. WITHIN

This relationship is many-to-many from the entity ACCESS PATH to the entity IKO. An access path is associated to an IKO if it belongs to the reference set of the IKO.

2.2.10. CONSISTS OF

This many-to-many relationship from the entity IKO to the entity ITEM indicates that a data item is one of the constituents of the IKO. The relationship has two properties :

ORD = A if the order is ascending on the data item.
    = D if the order is descending on the data item.
    = N if the IKO is not an order.

SEQ is the sequence number. If there is more than one item, the sort key is sorted on the sequence number of its data items (ascending).

2.2.11. TRANSFORMATION OF

This one-to-many relationship from the SCHEMA entity to itself is used to denote the fact that one schema is the transformation of another schema. The relationship has a property named TYPE_T. The property gives the type of transformation.

2.2.12. WORKS ON

This one-to-many relationship relates the ALGORITHM entity to the SCHEMA entity. It is used to express the fact that an algorithm works on a schema. The relationship is mandatory for the algorithm.

2.2.13. DERIVED FROM

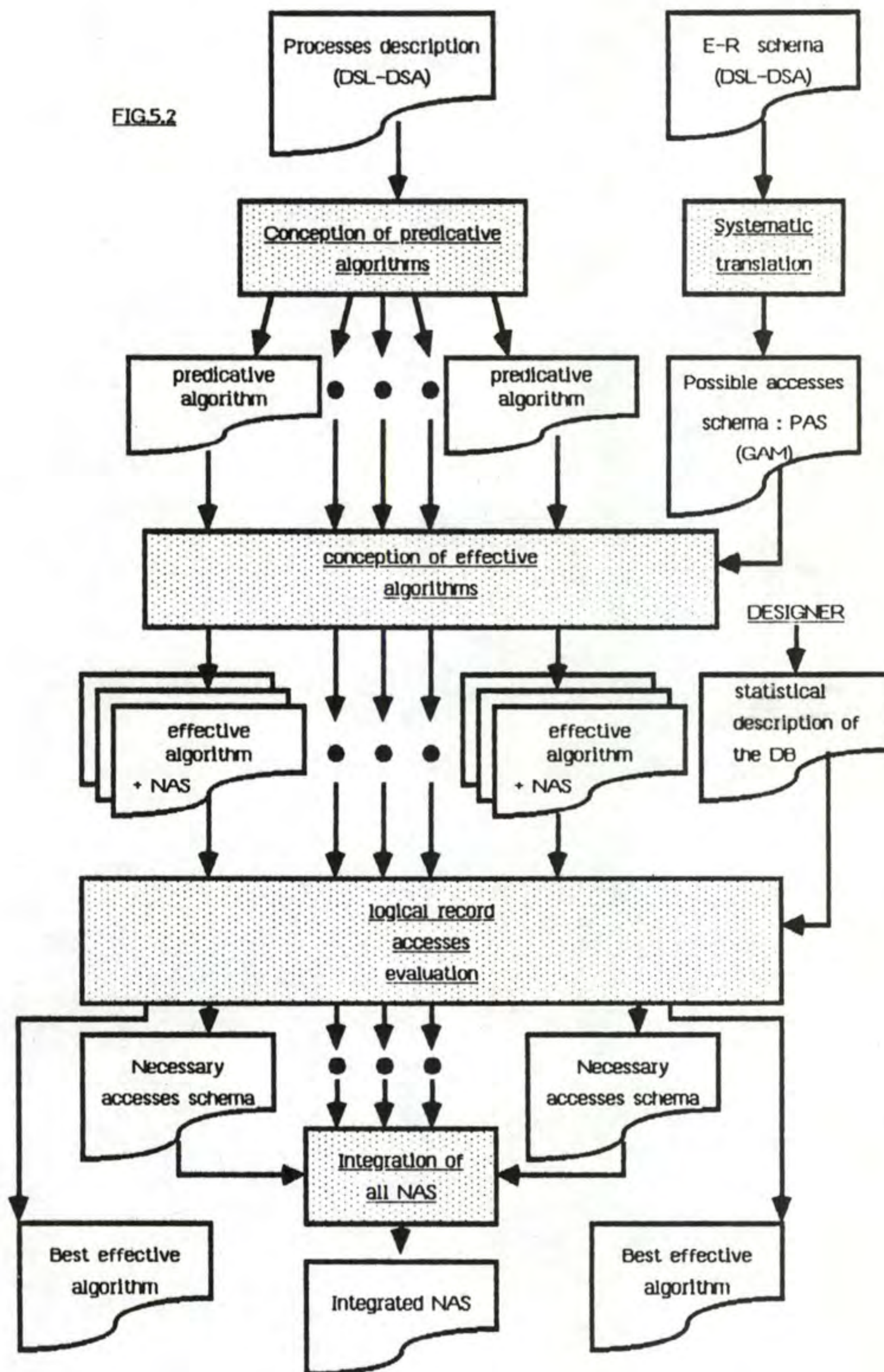
This one-to-many relationship relates the ALGORITHM entity to itself. It is used to denote the fact that one algorithm is derived from another one. The relationship has one property named TYPE_D. The property gives the type of derivation.

2.2.14. INVERSE OF

This one-to-one relationship from the entity ACCESS PATH to itself is used to represent the fact that one access path is the inverse of another one.

3. DESCRIPTION OF THE STEPS.

Figure 5.2 shows the mains steps of the suggested design process.



3.1. Systematic translation.

This step performs the translation of the data structures expressed with the E-R model, into a Possible Accesses Schema (PAS) expressed with the Generalized Access Model (GAM).

The PAS allows one to perform all possible accesses to the data structure -each data item becomes a key, each access path type has an inverse.

The GAM offers an expression of the data in a convenient way for the logical design -record type, item, access path type, access key.

This step could be automated.

3.2. Conception of predicative algorithms.

During this step, the designer has to write predicative algorithms. He uses the specifications of the processes (form the conceptual layer) as inputs.

Predicative algorithms specifies the manipulations of database informations without speaking about a particular sequence of accesses. They describe collections of database informations by means of their properties, and not by the way they can be accessed.

Predicative algorithms work on the data structures of the PAS, and are written in ADL.(access Algorithm Description Language).

3.3. Conception of effective algorithms and derivation of the corresponding NAS's.

During this step, for each predicative algorithm, the designer can obtain several effective algorithms and selects the corresponding NAS's.

An effective algorithm specifies the sequence of accesses that is uses to access the data. Each predicate is splitted in atomic actions, which are sequenced following a particular access strategy. An atomic action correspond to a database construct of the target DBMS. Therefore, the effective character of an algorithm is highly related to the class of the target DBMS.

In case there are many access strategies there will be many effective algorithms corresponding to one predicative algorithm.

A NAS is a restriction of the PAS, to the data structures needed by the effective algorithm which is associated with this NAS.

The output of this step is a set of pairs (NAS, effective algorithm) for each predicative algorithm.

3.4. Logical record access evaluation.

During this step, the designer use the Logical Record Access Evaluator to find the best pair (NAS, effective algorithm) out of each set of pairs.

The criteria is 'the least number of Logical Record Accesses (LRA)'.

The output of this step is an effective algorithm and the corresponding NAS, for each predicative algorithm.

3.5. Integration of all NAS.

Each NAS defined at the previous step is a particular restriction of the PAS. It is necessary to integrate all these schemas in one INTEGRATED NAS. This step could be automated.

4. An example.

4.1. Assumptions.

We assume that the conceptual design is already done. Thus, we have a conceptual schema to expresss the data structure (figure 5.3), an informal description of the processes, and we need a statistical description of the data which are going to fill the database.

The processes consists of two queries :

1. For a given name of product, list the name of the customers who send in an order for this product.

2. For a given 'N-ORD', display all the information about the order concerned :

   - N-ORD, DATE
   - NAME, N-CUS of the customer who send in the order
   - QUANT, N-PRO of the ordered products.

The statistical description reveals that there are :

20 000	customers	1 000	orders
1 000	products	3 000	lines of order

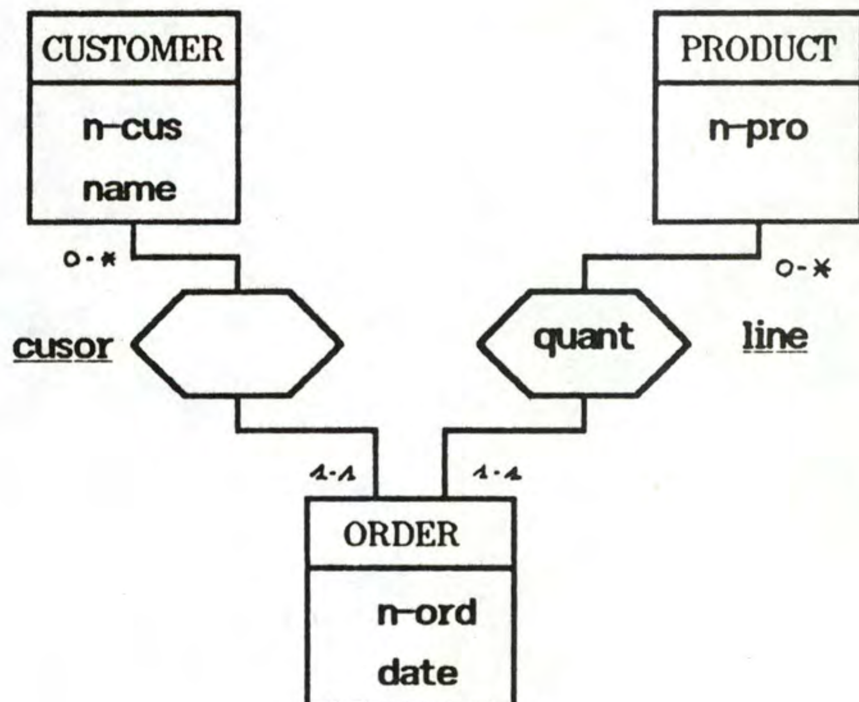


figure 5.3

4.2. First step : definition of the PAS (Possible Accesses Schema).

We have to define, with the GAM (Generelized Access Model), a data structure which is the exact image of the conceptual schema (E-R schema). We want to translate the end user view in a form which is more convenient for the process design of the logical layer. In terms of semantics, the PAS could be viewed as a conceptual schema.

The derivation of the PAS is systematic :

An ENTITY becomes a RECORD TYPE,

an ATTRIBUTE becomes an ITEM,

a RELATIONSHIP without attributes becomes an ACCES PATH TYPE,

a relationship with attributes becomes :

- a RECORD TYPE, with
- ITEM's, and
- at least two ACCESS PATH TYPE's (depending on the number of attributes).

The graphic representation of the PAS for the current example, is shown in figure 5.4

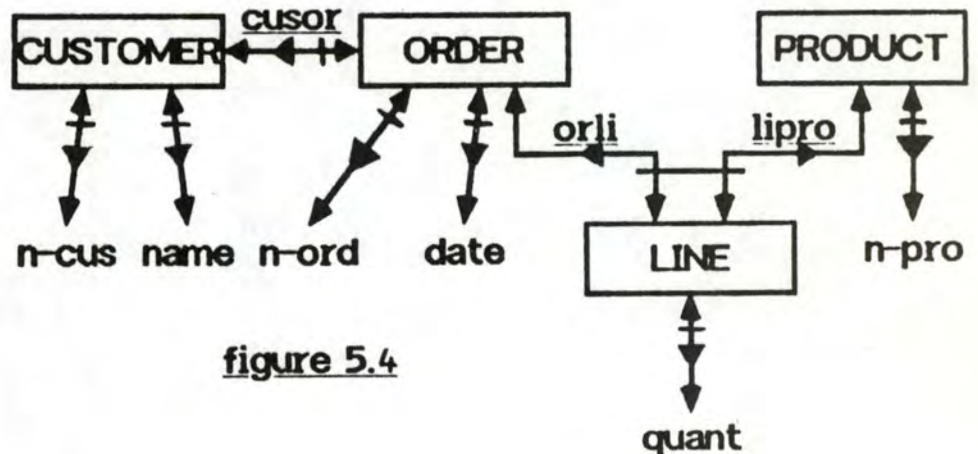


figure 5.4

4.3. Second step : Definition of the predicative algorithms.

For each process defined at the conceptual layer, the designer is alowed to use ADL to write a predicative algorithm. Every particular set of data which come from the PAS-- either a set af records, or a set of data item values, is indicated by :

- The name of the reference set-- for example : CUSTOMER, or LINE.
- The predicate that defines if an element of the reference set will be selected or not.

The first query might look like :

```
input(xn);
for cus :=
    CUSTOMER(CUSOR:ORDER(ORLI:LINE(LIPRO:PRODUCT(:N_PRO = XN))))
do
        print(name(:cus));
endfor;
```

4.4. Third step : Definition and selection of the effective algorithms.

At this level, the designer has to choose an effecient effective algorithm corresponding to the predicative algorithm that he wrote at the previous step. For one predicative algorithm, there are many effective algorithms, each one using a different sequence of data accesses.

Here is a non exhaustive list of effective algorithms corresponding to the first predicative algorithm :

ALG 1 :

```
input(XN);

for     CUS := CUSTOMER
do
        for     O := ORDER(CUSOR : CUS)
        do
                for     I := LINE(ORLI : O)
                do
                        P := PRODUCT(LIPRO : I);
                        if N-PRO( : P) = XN

                        then    print(NAME( :CUS));
                                next CUS;
                        endif;
                endfor;
                endif;
        endfor;
endfor;
```

ALG 2 :

```
input(XN);
for     I:= LINE
do
        if N-PRO( : PRODUCT(LIPRO : L)) = XN

        then    O := ORDER(ORLI : I);
                CUS := CUSTOMER(CUSOR : O);
                addlist(NAME( : CUS), LIST_NAME);
        ENDIF
ENDFOR;
uniqlist(LIST_NAME);
print(LIST_NAME);
```

ALG 3 :

```
input(XN);
P := PRODUCT( : N-PRO = XN);
for     L := LINE(LIPRO : P)
do
        O := ORDER(ORLI : L);
        CUS := CUSTOMER(CUSOR : O)
        addlist(NAME( : CUS)), LIST_NAME)
ENDFOR;
uniqlist(LIST_NAME);
print(LIST_NAME);
```

One way to choose between these effective algorithms is to evaluate for each, the mean value of logical accesses to the database. The figure 5.5 shows what should be the results of the evaluation if one considers the statistical description given in section 5.1.

	CUSTOMER	ORDER	LINE	PRODUCT	TOTAL
ALG 1	20 000	1 000	3000	3000	27 000
ALG 2	3	3	3000	3000	6 006
ALG 3	3	3	3	1	10

figure 5.5

If we wanted to optimize the number of LRA's, we should probably choose ALG 3. The NAS corresponding to ALG 3 is given in fig. 5.6.

The same process should be applied to the second predicative algorithm.

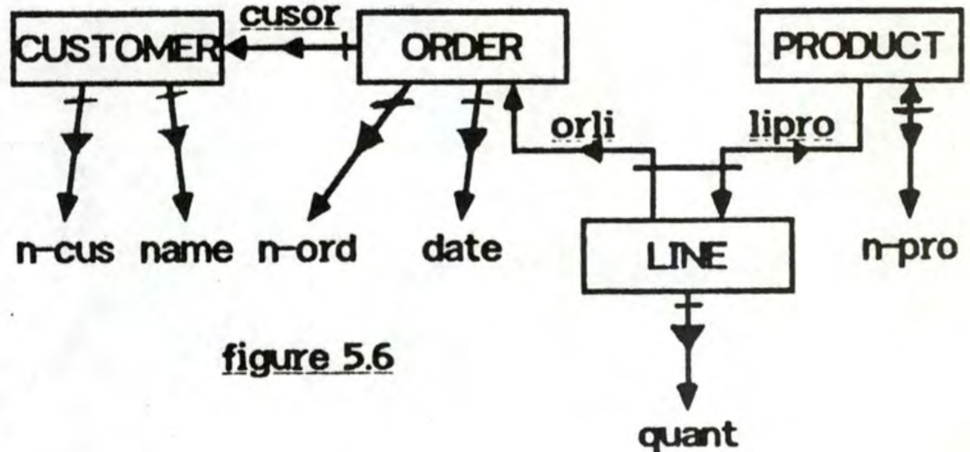


figure 5.6

4.5. Fourth step : Definition of the integrated NAS (Necessary Accesses Schema).

So far, we have selected an effective algorithm for the first query, an an effective algorithm for the second query. We have also two NAS's, which represent the data structures that the queries require. The integration of the two NAS's in one schema will give us the 'smaller' data structure we can implement.

Chapter 6: INTRODUCTION TO ISLDM/SEM

Introduction.

In this chapter are discussed the different concepts and goals of the ISDOS software. The first section will introduce the notions of Life cycle Support System (LSS), Information Proccesssing System (IPS), LSS processor, and LSS generator. The second section is concerned with a particular LSS generator, and processor, namely, the ISDOS software -ISLDM/SEM.

1. Life cycle Support Systems and Information Processing Systems.

As defined in [YUZO.81], a LSS is

"a computer based system that supports activities of a system department in one or more phases of the software life-cycle ".

The major functions of a LSS are described as the ability to :

- accept system description in some predefined notation.
- maintain a data base containing the system description.
- produce documentation and other outputs based on the system description.
- perform control functions of life-cycle activities.

The suggested methodology associated with the LSS to support software development is decomposed in two phases : the LSS generation and the LSS usage to describe an information system. This methodology is depicted in figure 6.1.

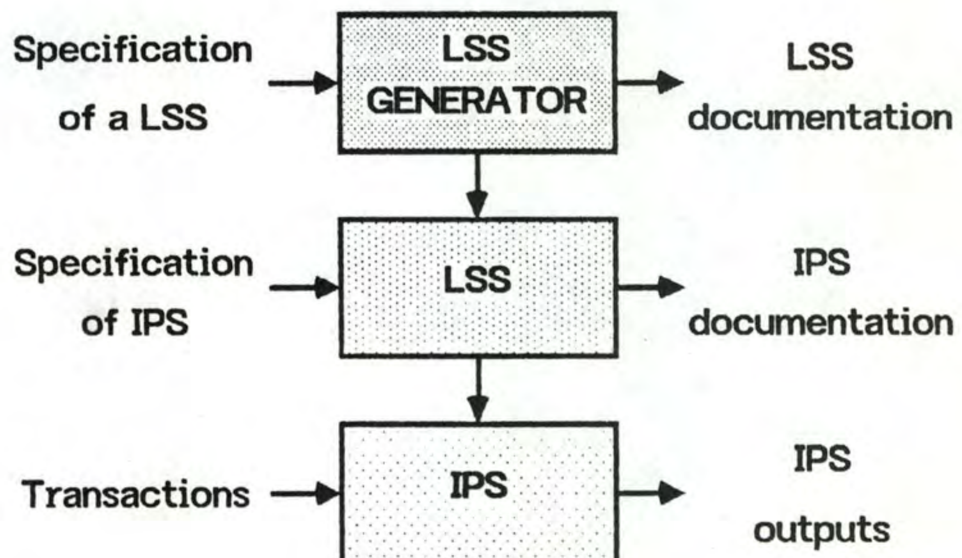


figure 6.1 from (YUZO.81)

A LSS generator is used to develop a specific LSS. The specific LSS characteristics are determined by the class of problems we want to describe using this LSS. The LSS generator receives as input the specific description expressed in the LSS description language and generates the corresponding LSS software and its documentation. Basically, the LSS software consists of :

- A language that allows the description of Information Processing System whose class has been predefined during the LSS definition.

- A set of tools allowing to maintain the IPS definition; that includes :

  - a processor for LSS language statements. The processor perform syntactical checks and checks for constistency against informations introduced in the data base about the system one want to describe.

  - an analyser to parse the content of the data base that contains a particular IPS description.

  - a set of tool that allows one to update the data base.

  - a documentation generator.

In turn, the LSS software is used to generate Information processing System (IPS). An IPS is defined in [TEICH.79] as

"the subsystem of the information system in which data is recorded

and processed following a formal procedure."

An IPS can be manual or computer-based. The term IPS will be used in this work to refer to computer-based system.

The overall data flow of an LSS is showm in figure 6.2. The IPS description is introduced in the LSS processor, in a predefined format, namely, the LSS language defined during the previous step, which is called Information System Description Language (ISDL). The LSS processor analyses those statements, performs checks and updates the LSS database. This database is a repository of informations describing the IPS; documentation and other reports are produced from informations contained in the data base.

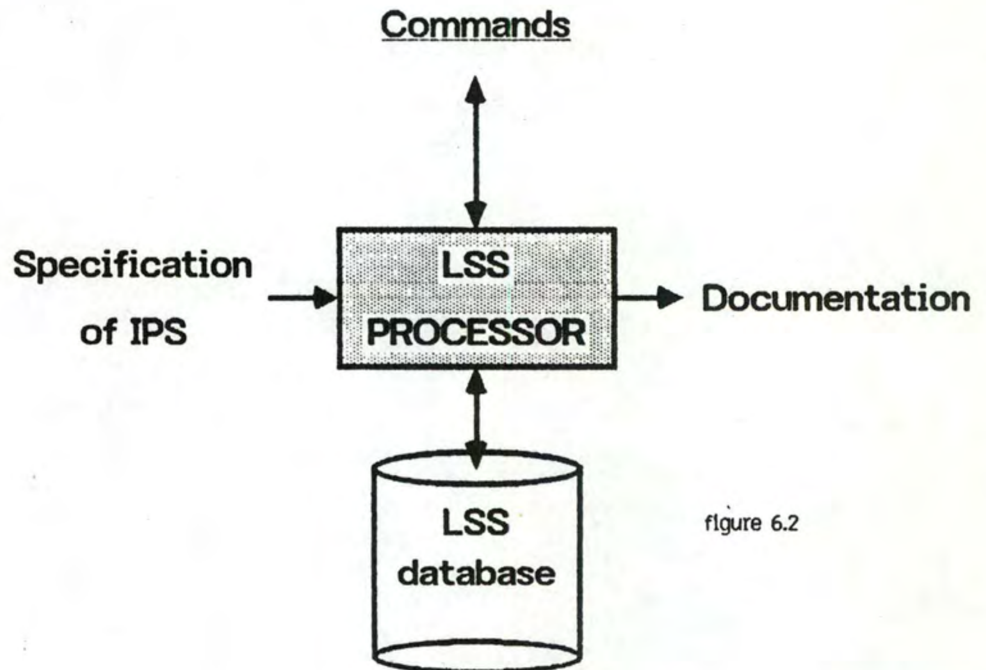


figure 6.2

2. ISLDM and SEM.

The Information System Language Definition Manager/System Encylopedia Manager (ISLDM/SEM) is an LSS generator example. The ISLDM/SEM structure is shown in figure 6.3. Following the methodology explained in the previous section, ISLDM/SEM is divided in two parts :

- the LSS definition part.

- the usage of the LSS, to define a particular IPS.

The first part consists in writting the ISDL description. The ISLDM (or META language) allows one to decribe an Information System Description Language. The META language can express information about an ISDL according to a model called META model. The META model used at ISDOS project is the E-R model. Thus, the statements describing a particular ISDL, and expressed with the META language, are analyzed by ISLDM. It checks that the language description is unambiguous, sintactically correct and consistent, and produces appropriate error messages, warnings, etc.. While analysing the language description, ISLDM updates a database containing the language definition. When the language designer is satisfied with his new ISDL definition, he can order the generation of tables containing the description of the language. ISLDM can also produce some reports, including the ISDL user's manual.

The second part consists in the use of the new language. The information system under development is described using the ISDL. The set of statements describing the IPS are introduced as text via a standard input device, or in the shape of a diagram via a graphics terminal. SEM receives those statements, analyzes them referencing the ISDL description contained in the tables generated by ISLDM. SEM also updates a database containing the Information System Description. This database is called the user-database. Every introduction of new data in the user database will be preceded by checks that will take in account both ISDL description, and current state of the database.

SEM is totally independent from a particular ISDL. The set of functions that SEM provides to the user does not change from one ISDL to another. The reference to a particular ISDL is achieve by means of tables generated during the ISDL definition phase.

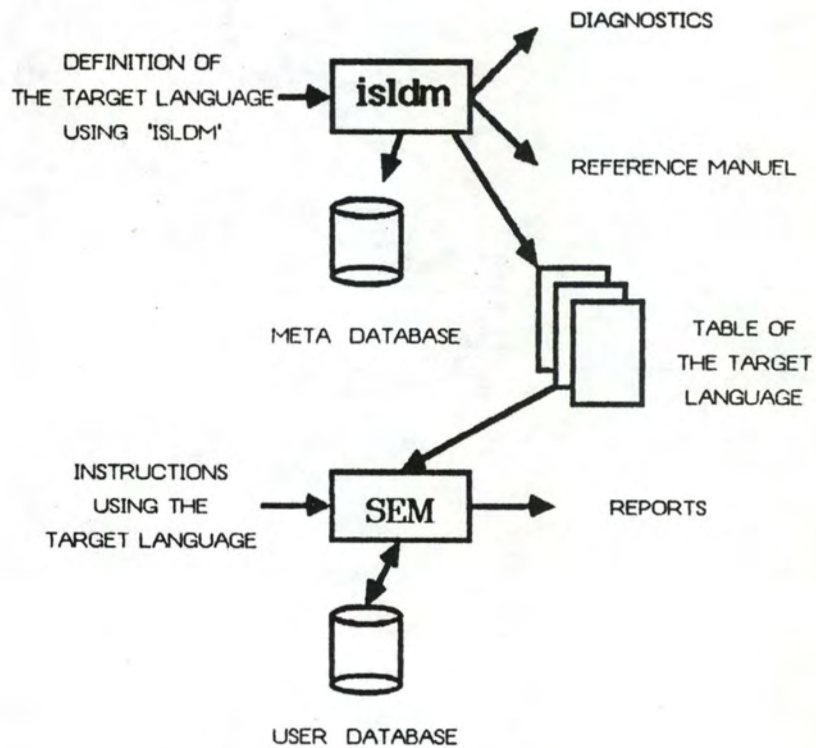


Figure 6.3

Among the functions provided by SEM, and in addition to the input processor previously described, we would mention the ability to :

- initialize the database containing the IPS description, before introducing it.

- update this database (IP: Input Processor, DP: Delete Processor, RP: Replace Processor).

- produce reports (FS: Formated Statements, STR: Structure, EP: Extended Picture).

- selects objects in the data base using NS (Name Selection) or the Query System.

Chapter 7: DESCRIPTION OF THE IMPLEMENTATION OF SDLA

1. Introduction

The Schema Description LAnguage (SDLA) was designed to describe all the information needed at the logical design layer. It is the implementation of the "global model" (see chapter 5). As such, it allows :

- the definition of logical database schemas based on the GAM model (see chapter 2).
- the definition of access algorithms written in ADL (see chapter 3).
- the statistical description of the database schemas. The description is based on the statistical model (see chapter 4).
- the expression of relationships between facts described by information belonging to the above information sets (schema descriptions, algorithms, statistical descriptions).

The above information is given in SDLA text form to the SEM input processor (see chapter 4) which feeds a "user database". Once the user database is build, the user can add, delete and modify the contents or produce reports.

SDLA is called, in ISDOS terminology, an Information System Description Language. The Description of SDLA is given in terms of the ISDOS Meta Model concepts.

We will give :

- the definition of the Meta Model concepts.
- a description of the SDLA language.
- the definition of the language in ISLDM (annexe [1]).

2. The Meta Model

The Meta Model supporting the SEM is essentially based on the Entity/Relationship model [CHEN.76], although different terminology is used. The three major concepts introduced in the E/R model (ENTITY, RELATIONSHIP, ATTRIBUTE) are included in this model with a fourth one added (CONSTANT). We will describe each concept in detail.

2.1. Object

An object is equivalent to the ENTITY notion in the E/R model. An object is a "thing which can be distinctly identified". It is the smallest unit that may be individually destroyed or created. For instance, the language designer can define the following objects : a file , a car, a database,... Each object has certain properties associated with it. Some of these properties are common to all objects in the language, others are introduced by the language designer and qualify specific objects.

2.2. Relationship

The relationship is an association between two or more objects or constants, different or not (up to 4). These components involved in the relationship are called "parts" of the relationship and the number of parts is called the "degree" of the relationship. The existence of the relationship depends on the existence of its components. Each instance of the relationship has the same degree. It is possible to limit the types of objects or constants that assume the role of a part in a specific relationship. One may also determine the "connectivity" of a relationship (see [BOD.83] p 26).

2.3. Property

A property is equivalent to the ATTRIBUTE notion in the E/R model. A property is a quality of an object or relationship and takes one or more values or value sets. We distinguish properties common to all objects from those that are introduced by the language designer.

The first class is called the "intrinsic properties". By definition these properties are predefined. Object name, date of last change of an object, degree of a relationship are examples of intrinsic properties.

The second set of properties are those defined by the language designer; they are dependent on the type of objects. For each property, the set of legal values is defined.

2.4. Constant

In addition to information about objects, the database contains constants. A constant is an object that represents itself and has no property. Some constants have a predefined type : integer number, real number, string. The language designer can define "name-constant" by giving a symbolic name to a constant value.

3. Description of the SDLA language

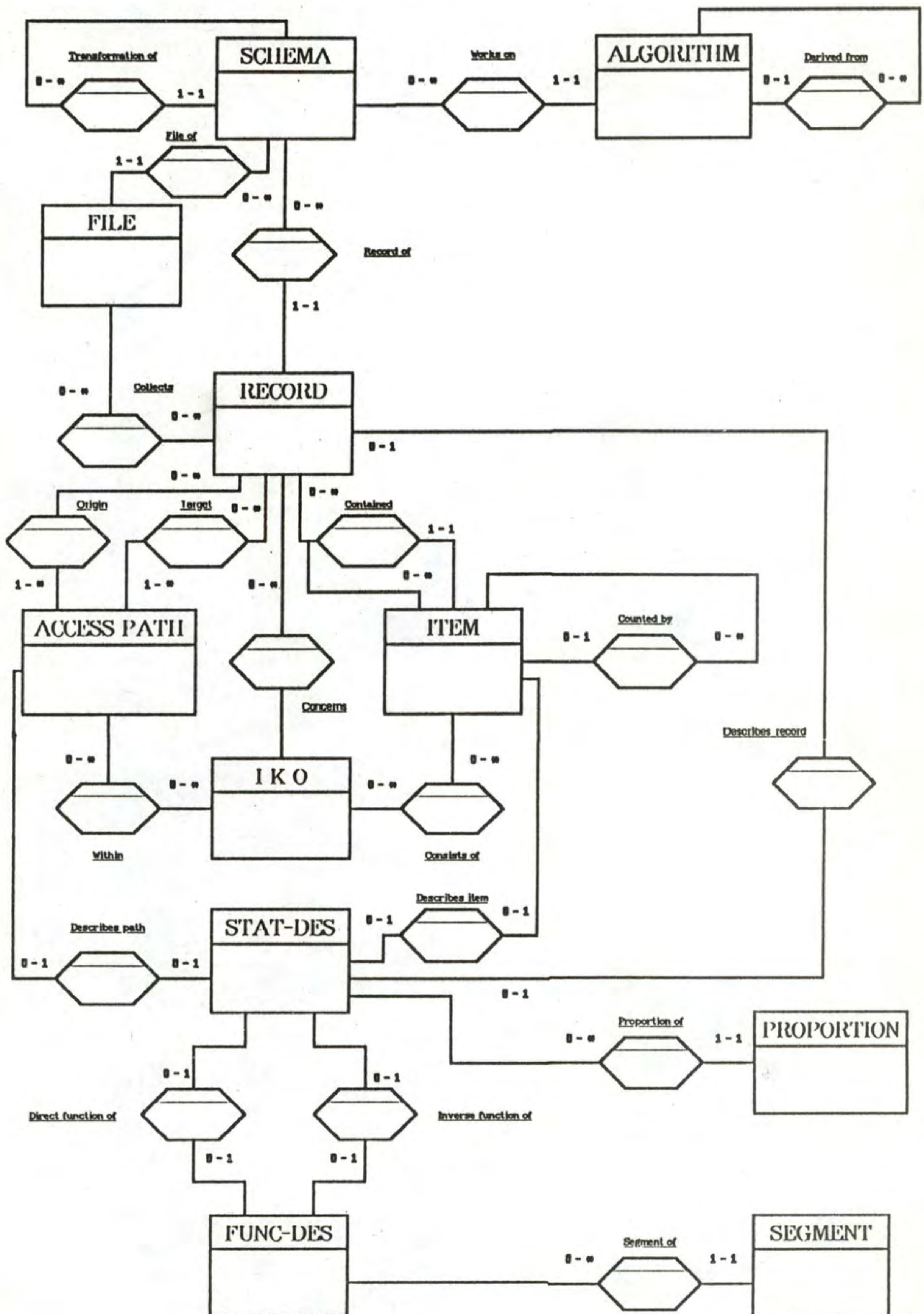
As the Meta Model supporting SEM is based on the E/R Model (see section 2), we will give first the E/R schema of the SDLA language. Later , we describe the language by giving for each object :

- a definition of the object.
- the properties associated to the object.
- the constants associated to the object.
- the description of the relationships in which the object takes part.
- The statements that can be used to associate the object to others by specified relationships.

The objects of SDLA are : SCHEMA, ALGORITHM, FILE, RECORD, ACCESS PATH, ITEM, IKO, STAT-DES, PROPORTION, FUNC-DES, SEGMENT.

Remq : Concepts which are particular to the methodology developed in Namur will be preceded by the "Methodology of Namur" paragraph heading.

E/R schema of the language.



3.1. SCHEMA

It represents a database schema expressed in the Generalised Access Model. A schema is a logical representation of data.

Methodology of Namur :

We consider different types of schema depending upon wether they are "close" to the conceptual schema or "close" to a particular DBMS approach to representing logical data (such as CODASYL or IMS schemas).

- The Possible Access Schema (PAS) :

  The PAS is the result of the systematic translation of the conceptual schema (expressed in the Entity/relationship model). Basically, it allows all accesses to the data. Each data item may be used as an access key, each access path type has an inverse.

- The Necessary Accesses Schema (NAS) :

  Usually, each algorithm does not need all the data structures specified in the PAS. The NAS is the restriction of the PAS to the accesses structures necessary to an algorithm. An "integrated NAS" is the result of the integration (union of access structures) of several NAS corresponding to an algorithm set. The integrated NAS is one of the outputs of the logical layer. It will be translated, at the physical design layer, into a particular DBMS schema.

Properties

- SCHEMA-TYPE : this property indicates the type of the schema.

  Methodology of Namur :

  As we consider only two types of schema, the property can take only two values : NAS and PAS.

Relationships

The object type SCHEMA is related by :

- a one-to-many relationship "WORKS-ON" to the object type ALGORITHM. It relates a SCHEMA with the access algorithms which works on it.
  The relationship is mandatory for the algorithm.

- a one-to-many relationship "RECORD-OF" to the object type RECORD. It associates a schema with the records belonging to it.

The relationship is mandatory for the record.

- a one-to-many relationship "FILE-OF" to the object type FILE. It means that the records belonging to one schema may be stored in several files. The relationship is mandatory for the file.

- a one-to-many relationship "TRANSFORMATION-OF" to the object type SCHEMA itself. It represents the fact that one schema is the transformation of another schema.

  Methodology of Namur :

  We saw that NAS are restrictions of PAS. This restriction is one type of schema transformation. Although one could imagine other types of transformation (conceptual schema to PAS, NAS to particular DBMS schema), we consider only one type of transformation : PAS -> NAS. TYPE-T is the property of the relationship "TRANSFORMATION-OF" which specifies the type of transformation. Value : PAS-NAS.

Statements

- TRANSFORMATION OF <schema-name>
  Specifies the schema from which a schema is the transformation.

3.2. ALGORITHM

It represents an access algorithm in ADL language. An algorithm works on the data structure specified by the schema related to it.

Methodology of Namur :

We distinguish two types of algorithms : predicative algorithms and effective algorithms.

- A predicative algorithm specifies collections of data by their properties and not by the way they can be accessed.

- An effective algorithm specifies the way it access to data.

A predicative algorithm is a detailled specification for effective algorithms. From one predicative algorithm, one could derive as many effective algorithms as they are ways to access the data structure.

Properties

- ALGORITHM-TYPE : this property indicates the type of the algorithm.

Methodology of Namur :

ALGORITHM-TYPE takes only two values : EFFECTIVE and PREDICATIVE.

Constants

- ALGORITHM-TEXT : It is a text constant. It contains the text of the algorithm in ADL. The input processor (IP) of SEM accepts texts as sequences of characters ended by a semicolon. No error checking of the ADL is made by SEM.

Relationships

The object type ALGORITHM is related by :

- a many-to-one relationship "DERIVED-FROM" to the object type itself. It denotes the fact that one algorithm is derived from another one.

  Methodology of Namur :

  It relates a predicative algorithm to the effective algorithms derived from it.
  The relationship is mandatory for effective algorithms.

- a many-to-one relationship "WORKS-ON" to the object type SCHEMA (see section 3.1).

Statements

- DERIVED FROM <algorithm-name>
  Used to specify the algorithm from which the algorithm is derived.

- WORKS ON <schema-name>
  Used to specify the schema on which the algorithm works.

3.3. FILE

It represents a GAM object FILE (see definition in chapter 2, section 2.4).

Relationships

The object type FILE is related by :

- a many-to-one relationship "FILE-OF" to the object type SCHEMA (see section 3.1).

- a many-to-many relationship "COLLECTS" to the object type RECORD. It relates a file to the record types belonging to it.

Statements

- FILE-OF <schema-name>
  Specifies the schema associated to a file by the FILE-OF relationship.

- COLLECTS <record-type-name>,...,<record-type-name>
  Used to specify the record types collected in a file.

3.4. RECORD

It represents a GAM object RECORD TYPE (see definition in chapter 2, section 2.1).

Relationships

The object type RECORD is related by :

- a many-to-many relationship "COLLECTS" to the object type FILE (see FILE section 3.3).

- a many-to-one relationship "RECORD-OF" to the object type SCHEMA (see SCHEMA section 3.1).

- a many-to-many relationship "ORIGIN" to the object type ACCESS PATH (see section 3.5). It relates a record type to the access path types of which it is origin. A record type can be origin of many access path types, an access path type can have many origin record types. The relationship is mandatory for the ACCESS PATH.

- a many-to-many relationship "TARGET" to the object type ACCESS PATH. It relates a record type to the access path types of which it is target. A record type can be target of many access path types, an access path type can have many target record types. The relationship is mandatory for the ACCESS PATH.

- a one-to-many relationship "CONTAINED" to the object type ITEM (see section 3.6). It relates a record type to its data items. Those are simple or decomposable data items. The relationship is mandatory for the ITEM.

- a one-to-one relationship "DESCRIBES RECORD" to the object type STAT-DES (see section 3.8). It relates a record with its statistical description.

Statements

- COLLECTED IN <file-name>
  Specifies the file to which the record type belongs.

- RECORD-OF <schema-name>
  Specifies the schema of which the record type is part.

3.5. ACCESS PATH

It represents a GAM object ACCESS PATH TYPE (see definition in chapter 2, section 2.3)

Properties

- FUNCTIONAL CLASS : It indicates the functional class of the access path type. This property takes four name-constant values :

  I-I : one origin, one target (one-to-one).
  I-M : one origin, many targets (one-to-many).
  M-I : many origins, one target (many-to-one).
  M-N : many origins, many targets (many-to-many).

Relationships

The object type ACCESS PATH is related by :

- a many-to-many relationship "ORIGIN" to the object type RECORD (see section 3.4).

- a many-to-many relationship "TARGET" to the object type RECORD (see section 3.4).

- a one-to-one relationship "INVERSE-OF" to itself. It relates an access path type to its inverse access path type.

- a many-to-many relationship "WITHIN" to the object type IKO (see section 3.7).

- a one-to-one relationship "DESCRIBES PATH" to the object type STAT-DES. It relates an access path type with its statistical description.

Statements

- INVERSE IS <access path name>
  Specifies the inverse access path type of the access path type.

- ORIGIN <record type name>,...,<record type name>
  Specifies the origin record types of the access path type.

- TARGET <record type name>,...,<record type name>
  Specifies the target record types of the access path type.

3.6. ITEM

It represents a GAM object DATA ITEM (see definition in chapter 2, section 2.2).

Properties

- OPTIONAL : The value of this property is YES if an occurrence of the associated record type or decomposable data item does not have to be always associated to a value of the data item. The value of the property is NO if the association between record type or decomposable data item occurrences and values of the data item is mandatory.

- REPETITIVITY : It indicates if the data item is repetitive or not. A data item is repetitive if more than one value of it can be associated to one occurrence of the associated record type or decomposable data item. The values taken by the repetitivity property are YES or NO.

Constants

- FORMAT : description of the data item format in COBOL way. FORMAT is a text constant.

  examples :

  999 : integer numbers from 0 to 999.
  X(5) : five alphanumeric characters.
  A(20) : 20 alphabetic characters.

Relationships

The object type ITEM is related by :

- a many-to-one relationship "CONTAINED" to the object type RECORD or to the object type ITEM itself. This is not a ternary relationship, it represents the fact that a data item is associated either to a record type or to a decomposable data item.

- a one-to-many relationship "COUNTED-BY" to the object type ITEM itself. It indicates that the data item is a counter of repetitivity for the associated data items.

- a many-to-many relationship "CONSISTS-OF" to the object type IKO (see section 3.7).

- a one-to-one relationship "DESCRIBES ITEM" to the object type STAT-DES (see section 3.8). It relates a data item to its statistical description.

Statements

- CONTAINED IN <data item name>|<record name>
  This statement is used to specify the record type or data item to which the data item belongs.

- COUNTED BY <data item name>
  It is used to specify the counter data item of a repetitive data item.

3.7. IKO

The IKO is the generalisation of three concepts : Identifier, access Key, Order. These concepts have been presented in the description of the GAM. Identifier is an integrity constraint (see chapter 2, section 4.2). Access key and order are GAM objects (see chapter 2, respectively section 2.6 and 2.7).

An occurrence of the object type IKO may express one, two or all these concepts at the same time.

Properties

- GLOBAL : Thru the concept of global, we want to extend the domain of validity of an identifier, an acces key , an order, to many record types. There are data items in different record types, which contains the same kind of information (for example WORKER-NUMBER and EMPLOYEE-NUMBER are both number of persons). If there is no intersection the set of possible values (WORKER-NUMBER takes values from 1 to 1000, and EMPLOYEE-NUMBER takes values from 1001 to 2000), we can say that both numbers are identifiers for both record types. In order to express this fact, we will say that both IKO belongs to the same global.

  A global is identified by an integer number called "global number". The value of the GLOBAL property is the number of the global to which the IKO belongs. If the IKO does not belong to a global, the value of the property is 0.

- REFERENCE-SET : it indicates the referential type of the IKO. We consider only three name-constant values for this property :
  - DB = all the database schema.
  - EF = each file taken separatly.
  - AP = the access paths related to the IKO.

- IDENT-KEY-ORDER : the value of this property is a string composed by 1 character minimum and 3 characters maximum. The allowed alphabet is ("I","K","O"). Although, the characters are facultative, they appear in the order "I","K","O". An "I", "K" or "O" is present depending upon whether the IKO represents respectively an identifier, an order, or a key.

- DUPLICATES : This property indicates if an acces key allows duplicates. The values of the property specify the order of storage of the duplicates elements :

  NO : no duplicates.
  LIFO : duplicates stored in LIFO order.
  FIFO : duplicates stored in FIFO order.
  RANDOM : duplicates stored in random order.

Constants

- ORDER-TYPE : this text constant describes, in free text format, the type of order.

Relationships

The object type IKO is related by :

- a many-to-many relationship to the object type ITEM. This relationship indicates that a data item is one of the constituents of the IKO. The relationship has two properties :

  ORD = A if the order is ascending on the data item.
  = D if the order is descending on the data item.
  = N if the IKO is not an order.

  SEQ is the sequence number. If there is more than one item, the sort key is sorted on the sequence number of its data items (ascending).

  example : last-name with SEQ = 1 and first-name with SEQ = 2, means that the record type people will be sorted first on the last-name and then on the first-name.

- a many-to-many relationship "WITHIN" to the object type ACCESS PATH. An access path is associated to the IKO if it belongs to the referential of the IKO.

- a many-to-one relationship "CONCERNS" to the object type RECORD. This relationship relates the IKO with the record type concerned by the identifier, the key or the order.

Statements

- CONSISTS OF <item-name> [WITH SEQ <seq-value> ORD <ord-value>],
  This statement is used to specify the data items constituent of an IKO.

- WITHIN <access path name>,...,<access path name>
  It is used to specify the access path types which constitute the referential of the IKO.

- CONCERNS <record-name>
  It is used to specify the record type concerned by the IKO.

3.8. STAT-DES

This object describes the statistical part of one occurrence of ACCESS PATH , RECORD or ITEM (see chapter 4).

Properties

- TYPE-P : if STAT-DES describes the relation between one complex object (record type or decomposable item) and one simple object (simple data item), TYPE-P gives the type of PROPORTION (see Statistical description).
  Values : real, integer, alphanumeric, alphanumeric with limited set of values.

- NUM-ELEMENTS : if STAT-DES describes a record type, it gives the number of occurrences of the record type in the database.

Relationships

The object type STAT-DES is related by :

- a one-to-one relationship "DESCRIBES-ITEM" to the object type ITEM. This relationship is used to relate a data item with its statistical description.

- a one-to-one relationship "DESCRIBES-PATH" to the object type ITEM. It is used to relate an access path type with its statistical description.

- a one-to-one relationship "DESCRIBES-RECORD" to the object type RECORD. It denotes the fact that one record type is statistically described by one STAT-DES object.

- a one-to-many relationship "PROPORTION-OF" to the object type PROPORTION. This relationship relates a STAT-DES of a simple data item to PROPORTION objects which describe the distribution of the data item values.
  The relationship is mandatory for the PROPORTION object.

- a one-to-one relationship "DIRECT-FUNCTION-OF" to the object type FUNC-DES. It relates a STAT-DES object with the description of the distribution of its direct function.

- a one-to-one relationship "INVERSE-FUNCTION-OF" to the object type FUNC-DES. It relates a STAT-DES object with the description of the distribution of its inverse function.

Statements

- DIRECT FUNCTION IS <func-des-name>
  Used to specify the direct function description FUNC-DES associated to a STAT-DES object.

- INVERSE FUNCTION IS <func-des-name>
  Used to specify the inverse function description FUNC-DES associated to a STAT-DES object.

- DESCRIBES PATH <path-name>
  Used to associate a STAT-DES object to the access path it describes.

- DESCRIBES ITEM <data-item-name>
  Used to associate a STAT-DES object to the data item it describes.

- DESCRIBES RECORD <record-name>
  Used to associate a STAT-DES object with the record type it describes.

3.9. FUNC-DES

This object type represents one distribution function description of a relation (see chapter 4). It collects statistical information about the direct or (excluding) inverse function of a relation.

Properties

- FUNCTION-TYPE : gives the type of the function (see chapter 4). Examples : REAL-FUNCTION, POISSON, BINOMIAL ... .

- MEAN : gives the average of the distribution function.

- F-ZERO : is the value of the distribution function in 0.

- K-MAX : gives the point where the function is maximum.

- PAR-ONE : parameter needed for certain special function types (ex : MAX1-FUNCTION see chapter 4).

- PAR-TWO : same as PAR-ONE

- MIN-RELATIONS : gives the value of ID or II (see chapter 4).

- MAX-RELATIONS : gives the value of JD or JI (see chapter 4).

- NUMBER-TARGET : if the described relation is R (origin A , target B), the property gives :
  - the number of B objects if FUNC-DES describes a direct function.
  - the number of A objects if FUNC-DES describes an inverse function.

Relationships

The object type FUNC-DES is related by :

- a one-to-one relationship "DIRECT-FUNCTION-OF" to the object type STAT-DES (see section 3.8).
- a one-to-one relationship "INVERSE-FUNCTION-OF" to the object type STAT-DES (see section 3.8).
- a one-to-many relationship "SEGMENT-OF" to the object type SEGMENT. It relates a FUNC-DES object of FUNCTION-TYPE = REAL-FUNCTION to the SEGMENT objects which describe it. The relationship is mandatory for the SEGMENT object type.

Statements

- SEGMENTS ARE <segment-name>,...,<segment-name>
  It is used to specify the SEGMENT objects which belongs to a function description.

3.10. SEGMENT

When the distribution function is a REAL-FUNCTION, we need a variable number of classes to describe this function. We use a SEGMENT object to describe one class.

Properties

- MAX-CLASS-S : gives the upper limit of the class.
- VALUE-S : gives the proportion of occurrences of A (B if inverse) related to n occurrences of B (A if inverse) with n included in the class.

Relationships

The SEGMENT object type is related by :

- a many-to-one relationship "SEGMENT-OF" to the object type FUNC-DES (see section 3.9).

Statements

- SEGMENT-OF <func-des-name>
  Used to specify the FUNC-DES object to which a segment belongs.

3.11. PROPORTION

An object of this type describes one class of the proportion function (see chapter 4). A proportion function is needed only when one wants to describe a relation involving a non decomposable data item. The function gives the distribution of data item values.

Properties

- MAX-CLASS-P : gives the upper limit of the class.
- VALUE-P : gives the proportion of occurrences of A which have a B value included in that class.

Relationships

The object type PROPORTION is related by :

- a many-to-one relationship "PROPORTION-OF" to the object type STAT-DES (see section 3.8).

Statements

- PROPORTION-OF <stat-des-name>
  Used to specify the STAT-DES object to which the PROPORTION object belongs.

Chapter 8: INTRODUCTION TO LEX AND YACC.

1. INTRODUCTION.

The development of new programs on the UNIX system is facilitated by tools for language design and implementation. These are frequently programs generators, compiling into C, which provide advanced algorithms in a convenient form .One of the most important such tools are Yacc, a generator of LALR(1) parsers , and Lex, a generator of regular exression recognisers using deterministic finite automata.

This chapter focuses on Yacc and Lex. A detailed description of the underlying theory of both programs can be found in [AHO.77], while the appropriate user's manual can be consulted for further examples and details [JOHNS.75] [LESK.75].

Since programs generators have output which is turn input to a compiler and the compiler output is a program which in turn may have both input and output, some terminology is essential. To clarify the discussion thoughout this chapter, the term 'specification' will be used to refer to the input of Yacc or Lex. The output program generated then becomes the 'source', which is compiled by the host language compiler. The resulting 'executable object' program may the read data and produce output. To use a generator :

- The user writes a specification for the generator, containing grammar rules (for Yacc) or regular expression (for Lex).
- The specification is fed through the generator to produce a source code file.
- The source code is processed by the compiler to produce an executable file.
- The user's real data is processed by the executable file to produce real output.

This can be diagrammed as shown in Fig 8.1. Both Yacc and Lex accept both C and Ratfor [KERN.75] as host language, although C is far more widely used.

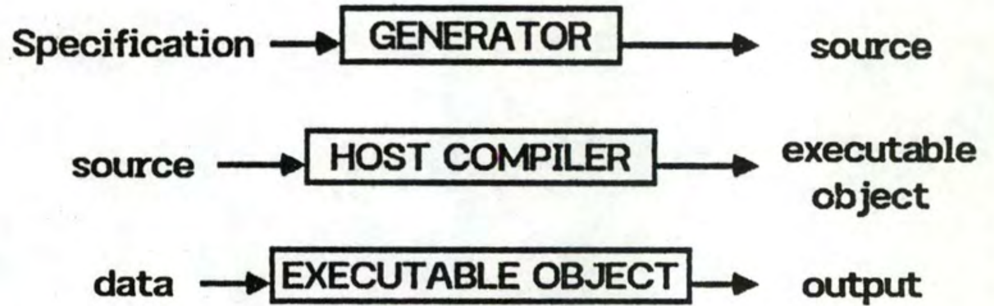


figure 8.1

The remainder of this chapter gives more detail on the two programs generators, Yacc in section 2 and Lex in section 3. Section 4 discusses the cooperation between YACC and LEX.

2. THE YACC PARSER GENERATOR.

Yacc is a tool which turn a wide class of context-free grammars (also known as Backus-Naur Form, or BNF, descriptions) into parsers that accept the language specified by the grammar. A simple example of such a decription might look like :

```
date :        month day year ;

month :       "JAN" | "FEB" | "MAR" |

              "APR" | "MAY" | "JUN" |

              "JUL" | "AUG" | "SEP" |

              "OCT" | "NOV" | "DEC" ;

day :         number ;

year :        "," number |      ;

number :      DIGIT |

              number DIGIT ;
```

In words, this says that a date is a month, day, and year, in that order; in Yacc style of writing BNF, colons and semicolons are syntactic connectives that aren't really included in the actual description. The vertical bar stands for "or", so a month is JAN or FEB and so on. Quoted strings can stand for literal appearances of the quoted characters. A day is just a number (discussed below). A year is either a comma followed by a number, or it can in fact be missing entirely. Thus, this example would allow as date either JUL 4,1776 or JUL4.

The two rules for number say that a number is either a single digit, or a number followed by a digit. Thus, in this formulation, the number 123 is made up of the number 12, followed by the digit 3; the number 12 is made up of the number 1 followed by the digit 2; and the number 1 is made up simply of the digit 1.

Using Yacc, an action can be associated with each of the BNF rules, to be performed upon recognising that rule. The actions can be any arbitrary program fragments. In general, some value or meaning is associated with the components of the rule, and part of the job of the action for a rule is to compute the value or meaning to be associated with the left side or the current rule. Thus, a mechanism has been provided for this program fragments to obtain the values of the components of the rule, and return a value. Using the possible digit; the value of ´1´ is 1, etc.. .The rules describing the structure of numbers can be followed by associated program fragments which compute the meaning or value of the numbers. Assuming that numbers are decimal, the value of a number which is a single digit is just the value of the digit, while the value of the number which is a number followed by a digit is 10 times the value of the number, plus the value of the digit. In order to specify the values of numbers, we can write :

```
number :        DIGIT

                {       $$=$1;  }

       |        number DIGIT

                {       $$= 10*$1 +$2;  }

       ;
```

Notice that the values of the components of the right-hand side of the rule are described by the ´pseudo-variables´ $1,$2,etc... which refer to the first, second, etc... elements of the right side of the rule. A value is returned for the rule by assigning to the ´pseudo-variable´ $$. After writing the above actions, the other rules wich use ´number´ will be able to acces the value of the number.

Recall that the values for the digits were assumed known. In practise, BNF is rarely used to describe the complete structure of the input. Usually a previous stage, the ´lexical analyser´, is responsible for actually reading the input characters and assembling them into ´tokens´, the basic input units for the BNF specification. Lex, described in the next section, is used to help build lexical analysers; among the issues usually dealt with in Lex specifications are the assembly of alphabetic characters into names, the recognition of classes of characters (such as DIGITS), and the treatment of blanks, newlines, comments, and other similar issues. In particular, the lexical analyser will be able to associate values to the tokens which it represents, and these values will be accessible in the BNF specification.

The programs generated by Yacc, called parsers, read the input data and associate the rules and actions of the BNF to this input, or report an error if there is no correct association. If the above BNF is given to Yacc, togheter with an appropriate lexical analyser, it will produce a program that will read dates and only dates, report an error if something is read that does not fit the BNF description of a date, and associate the correct actions, values, or meanings to the structure encountered during input.

It is also possible to enclose the Yacc-generated parser in a larger program. Yacc translate the user's specification into a programm named 'YYPARSE'. This program behaves like a finite automaton that recognizes the user's grammar; it is represented by a set of tables and an interpreter to proccess them. If the user does not supply an explicit main program, 'YYPARSE' is invoked and it reads and parse the input sequence delivered by the lexical analyser. If the user whishes, however, a main program can be supplied to perform any desired actions before or after calling the parser, and the parser may be invoked repeatedly. The function value returned by 'YYPARSE', indicates whether or not a legal sentence in the specified language was recognised.

It is also possible for the user to introduce his own code at a lower level, since the Yacc parser depends on a routine 'YYLEX' for its input and lexical analysis. This subroutine may be written with Lex (see the next section) or directly by the user. In either case, each time it is called, it must return a lexical token name to the Yacc parser. It can also assign a value to the current token by assigning to the variable 'yyval'. Such values are used in the same way as values assigned to the $$ variables in parsing actions.

Thus the user'code may be placed (i)above the parser, in the main program ; (ii) in the parser, as actions statements on rules; and/or (iii) below the parser, in the lexical analyser. All of these are in the same core load, so they may communicate through external variables as desired. Note , however, that despite the presence of user code even within the parser, both the finite automaton tables and the interpreter are entirely under the control of, and generated by, Yacc, so that changes in the automaton representation need not affect the user.

In addition to generality, the time required for Yacc to process most specifications is faster than the time required to compile the resulting C programs. The parsers are also comparable in space and time with those that may be produced by hand, but seem to be much easier to write and modify, especially by non specialists.

To summarize, Yacc provide a tool for turning a wide class of BNF descriptions into efficient parsers. It is packaged as a program generator, and requires a lexical analyser to be supplied. The next section will discuss a complementary tool, Lex which builds lexical analyser suitable for Yacc.

3. THE LEX LEXICAL ANALYSER GENERATOR.

Lex is similar in spirit to Yacc, and there are many similarities in its input format as well. Like Yacc, Lex input consists of rules and associated actions. Like Yacc, when a rule is recognised, the action is performed. The major differences arise from the typical input data and the model used to process them. Yacc is prepared to recognize BNF rules on input which is made up of tokens. These tokens may represents several inputs characters, such as names or numbers, and there may be characters int the input text that are never seen by the BNF description (such as blanks). Programs generated by Lex on the other hand, are designed to read the input characters directly. The model implemented by Lex is more powerful then Yacc at dealing with local information -contaxt, character, classes and repetition - but is almost totaly lacking in more global structuring facilities such as recursion. The basic model is that of the theory of regular expressions, which also underlies the UNIX text editor 'ed' and a number of other UNIX programs that ptocess text. The class of rules is choosen so that Lex can generate a program that is a deterministic finite state automaton : this means that the resulting analyser is quite fast, even for large sets of regular expressions. The program fragments written by the user are executed in the order in which the corresponding regular expressions are matched in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substancial look-ahead is performed on the input, but the input stream will be backed up to the end on the final string matched, making this look-ahead invisible to the user.

As with Yacc, Lex actions may be general program fragments. Since the input is believed to be text, a character array (called 'yytext') can be used to hold the string which was matched by the rule. Actions can obtain the actual characters matched by accessing this array.

The structure of Lex output is similar to that of Yacc. A function named 'YYLEX' is produced, which contains tables, and an interpreter representing a deterministic finite automaton. By default, 'YYLEX' is invoked from the main program., and it reads characters from the standard input. The user may provide his own main program however. Alternatively, when Yacc is used, it automatically generates calls to 'YYLEX' to obtain input tokens. In this case, each Lex rule which recognizes a token should have an action

return ('token-number')

to signal the kind of token recognized to the parser. It may also assigned a value to 'yyval' if desired.

The user can also change the Lex input routines, so long as it is remembered that Lex expects to be able to look ahead on and then back up the input stream. Thus, as with Yacc, user code may be above, within, and below the Lex automaton. It is even easy to have a lexical analyser in which some tokens are recognized by the automaton, and some by user written code. This may be necessary when somme input structure is not easily specified by the class of regular expressions supported by Lex.

The definition of regular expressions are very similar to those in QED [KERN.72], and UNIX text editor ´ed´ [THOMPS.75]. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and the operators (which specify repetitions, choices, and other features). The letters of alphabet and the digits are always text characters ; thus the regular expression

integer

matches the string ´integer´ wherever it appears and the expression

a57D

looks for the string ´a57D´. It is also possible to use the standard C language escapes to refer to certain special characters, such as \n for newline and \t for tab.

4. COOPERATION OF YACC AND LEX.

Since Yacc generates parsers and Lex can be used to make lexical analysers, it is often desirable to use them togheter to make the first stage of a language analyser. In such an application, two specifications are needed :

- A set of lexical rules to define the input data tokens.

- a set of grammar rules to define how these tokens may appear in the language.

The input data text is read, divided up into tokens by the lexical analyser, and then, passed to the parser and organized into larger structures of the input language. In principles this could be done with pipes, but usually the code produced by Lex and Yacc are compiled together to produce one program for execution. Conventionnally, the Yacc programm is named 'YYPARSE', and it calls output source program. The overall appearance is shown in Fig 8.2.

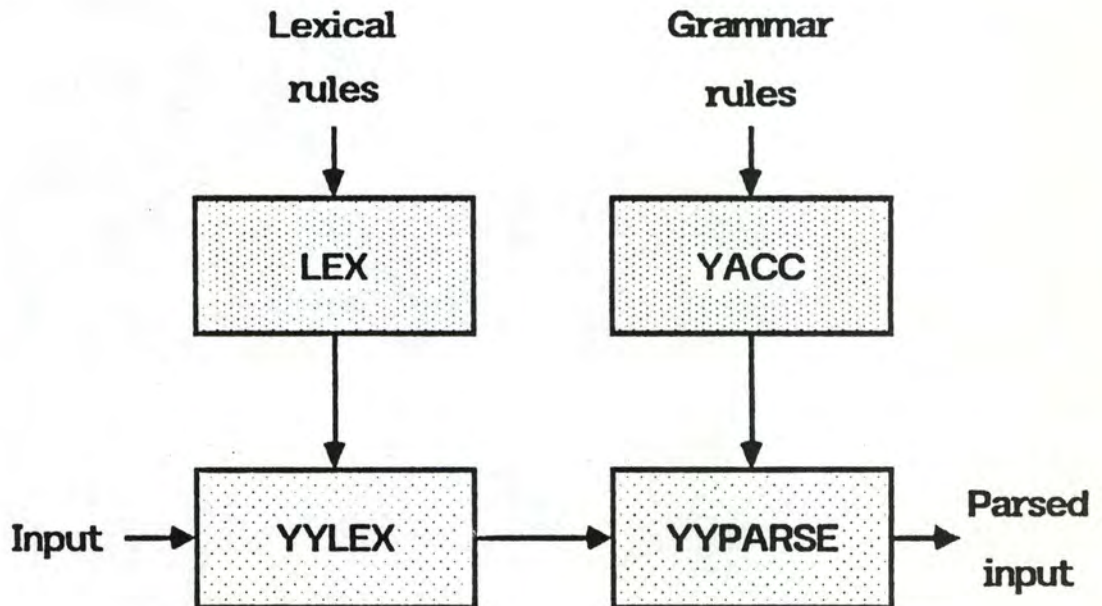


figure 8.2

To make this cooperation work, it necessary for Yacc and Lex to agree on the numeric code used to differenciate token types. These codes can be specified by the user, but ordinarily the user allows Yacc to choose these numbers, and Lex obtains the values by including a header file written by Yacc, which contains the definitions. It is also necessary to provide a mechanism by which Yacc can obtain the values of tokens returned from Lex. These values are passed through the external variable ´yylval´.

Yacc and Lex were designed to work together, and are frequently used together. The programs using this technology include the portable C compiler and the C language preprocessor.

Chapter 9: DESIGN AND IMPLEMENTATION OF AN ADL ANALYSER.

SUMMARY.

This chapter presents the choices that was made to design the ADL analyser. The ADL analyser accepts a source text written in ADL, and produces a parse tree and a symbol table.

The first section concerns the tree structure. It presents the components of a node, and the links between them. Besides, it gives a description of the routines that allow one to manipulate the tree. Finally, it defines what is a parse tree.

The second section concerns the symbol table. It gives the structure of the table. Moreover, it describes the routines that allows one to access the symbol table.

The activities of the analyser may be decomposed in a LEXICAL part (section 3), a SYNTACTICAL part (section 4), and a SEMANTIC part (section 5). Each part was designed separatly. We believe that separation of concerns and information hiding can help to specify programs easy to write, to maintain, and to understand. However, we are conscious that it is not the best way to achieve the faster analyser, but since we are in an experimental phase, we will focus on the ease to write, to maintain and to understand, prior to performances in terms of response time and memory space.

1. THE TREE STRUCTURE.

1.1. Ordinary trees.

Generally speaking, tree structure means a 'branching' relationship between nodes, much like that found in the trees of nature. [KNUTH.81] defines a tree as

" A finite set T of one or more nodes such that :

1) There is one specially designated node called the root of the tree, and

2) The remaining nodes (excluding the root) are partionned into m >= 0 disjoints sets T1, ... , Tm, and each of these sets in turn is a tree. The trees T1, ... , Tm are called subtree of the root. "

Standard terminology for the tree structures is taken from a form of family tree, the lineal chart :

- Each root is said to be the FATHER of the roots of its subtrees,

- the latter are said to be the SONS of their father,

- the sons of the same father are called BROTHERS.

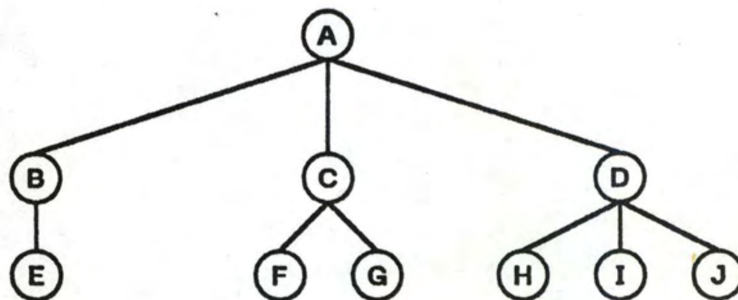
Remarks :

- The root of the entire tree has no father.

- Terminal nodes have no son.

- Non terminal nodes have at least one son.

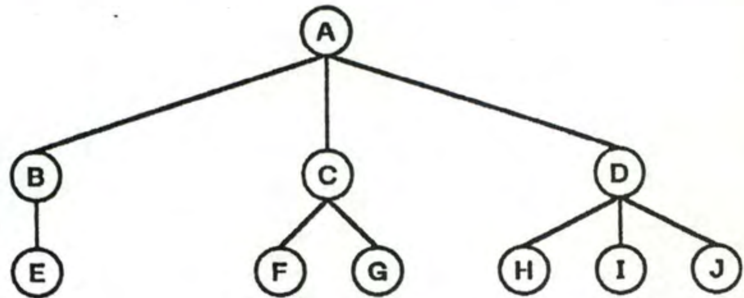
Example.



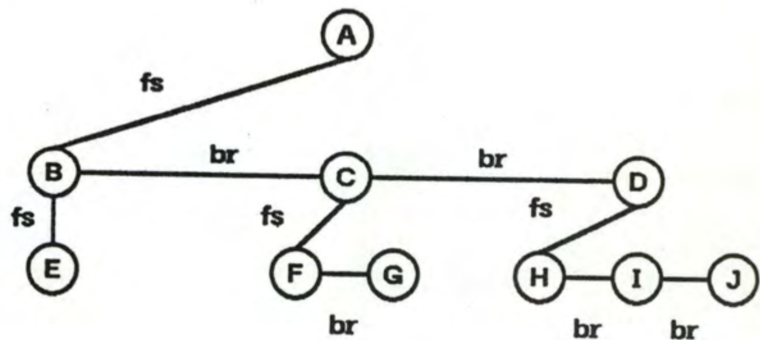
1.2. Binary trees.

A particular kind of tree is the binary tree. A binary tree has a finite set of nodes which either is empty, or consists of a root and two disjoints binary trees called the left and right subtree of the root.

There is a natural way to represent an ordinary tree by a binary tree. Consider the following ordinary tree



The corresponding binary tree is obtained by linking together the sons of each family and removing vertical links except from a father to his first son.



<u>Implementation of an ordinary tree, using a binary tree structure.</u>

Many applications of tree structure requires rather frequent references upward in the tree as well as downward. In order to facilitate "traversing", the structure of the node is

INFO	FATHER
SON	BROTHER

where :

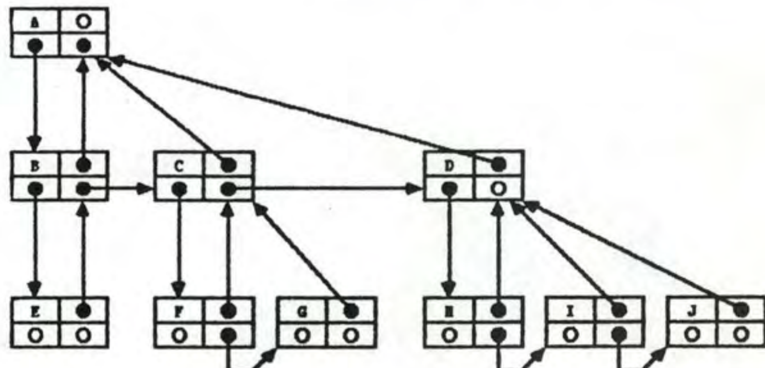
- INFO is a field dedicated to the user. Its structure depends on what informations the user want to bind with a node.

- FATHER contains the reference of the node which is the father of the described node. As far as the described node has no father, the value of its field FATHER will be NUL(NUL is diagrammed as ´o´).

- SON contains the reference of the node which is the first son of the described node. As far as the described node has no son, the value of its field SON will be NUL(NUL is diagrammed as ´o´).

- BROTHER contains the reference of the node which is the brother of the described node. As far as the described node has no brother, the value of its field BROTHER will be NUL(NUL is diagrammed as ´o´).

and the tree, which is called a TRIPLY LINKED TREE, look like



1.3. Routines that work on binary trees.

These routines work on a single triply linked tree. Moreover, the information part consists of two subparts : one contains the type of the node, and the other contains the identifier of the node. Thus the node consists of :

- a field named "ID" which contains the identifier of the node. If one want to add information to the node, we suggest him to use the "ID" as entry in a indirection table that will contains the reference of the information associated with the node.

- a field named "TP" which contains the type of the node.

- a field named "FA" which contains the reference of the father of the node.

  a field named "SO" which contains the reference of the first son of the node.

- a field named "BR" which contains the reference of the brother of the node.

For each field, we provide a routine to read the field, and a routine to update the field. Besides, we provide some utility routines.

READ/WRITE routines.

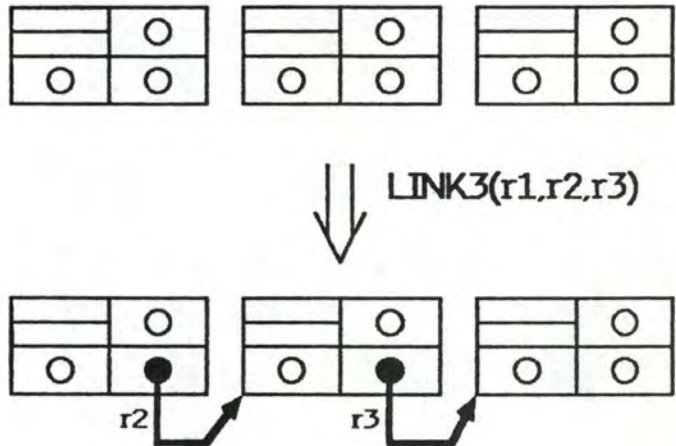
- IDENT(r), TYPE(r), FATHER(r), BROTHER(r), SON(r) give the value of the corresponding field. ´r´ is the reference of the node.

- CIDENT(r,v), CTYPE(r,v), CFATHER(r,v), CBROTHER(r,v), CSON(r) allows one to update the corresponding field. ´r´ is the reference of the node, ´v´ is the updating value.

<u>UTILITY</u> <u>routines</u>.

- ROOT( ) gives the reference of the root of the entire tree.

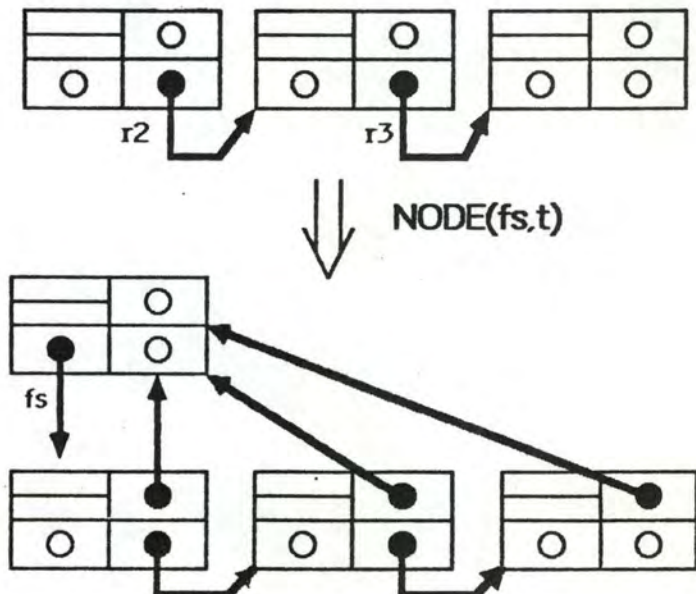
- LINK2(r1,r2), LINK3(r1,r2,r3), LINK4(r1,r2,r3,r4), LINK5(r1,r2,r3,r4,r5), allow one to link several node as 'BROTHER'.



- NODE(t,fs) allows one to create a node of type 't'. However the routine does much than simply create a node. It accepts the reference of the first son, namely 'fs', and update the field FATHER of the first son and all the brothers of the latter.

<u>Remark</u> :

We assume that all the sons of the same family are created and linked together before the creation of the father. Therefore, the creation of a binary tree, using these routines must be done 'bottom up'.



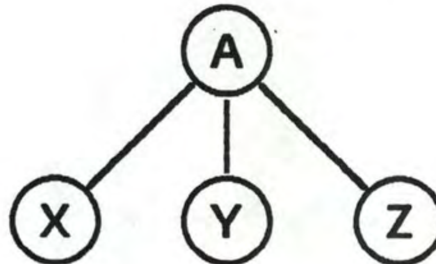
1.4. A particular use of trees, the parse trees.

Parse trees have the important purpose of making explicit the hierachical syntactic structure of sentences that is implied by the grammar.

Each interior node of the parse tree is labeled by some nonterminal A, and the childrens of the node are labeled, from left to right, by the symbols in the right side of the production by which A was replaced in the derivation. For example, if

A -> XYZ

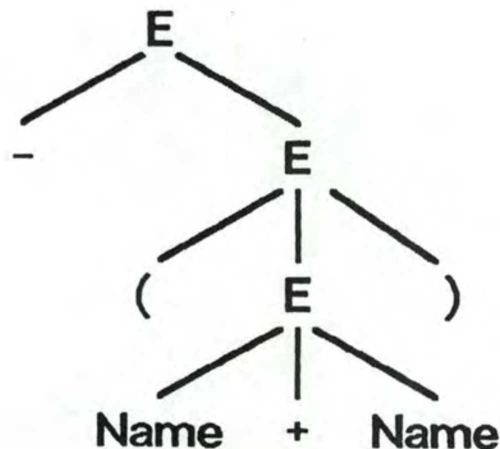
is a production used at some step of a derivation, the parse tree for that derivation will have the subtree



the leaves of the parse tree are labeled by nonterminals or terminals and, read from left to right, they constitute a sequential form called the yield of the tree. For example, the parse tree for -(name+name) implied by the following derivations

E => -E => -(E) => -(E+E) => -(name + E) => -(name + name)

look like



2. THE SYMBOL TABLE.

Introduction

The parser needs to collect information about all the data objects that appear in the source program. For example, the semantic analyser needs to know whether a variable represents an integer or a real number, what size an array has, how many arguments a function expects, and so forth. These information is entered in a data structure called symbol table. In the abstract, a symbol table is merely a table with two fields, a name field and an information field. The names in the symbol table denote objects of various sorts. There may be variables names, functions names, etc ... We require several capabilities of the symbol table. We need to be able to

- Determine whether a given name is in the table,
- Add a new name in the table,
- Access the information associated with a given name,
- Add new information for a given name,
- Remove a name from the table.

The primary issues in symbol table design are the format of the entries, the access method.

The first section presents the data structure of the symbol table, while the second section lists the routines that access the symbol table.

2.1. Data structure.

Each entry of the symbol table consists of :

- the string which is the name of the declared object.
- the length of the string.
- the class of the name.
- the reference to the declaration tree.
- the level number.

The class of the name determines which type of object the name refers to. The class is either schema, algorithm, function, or variable.

Each object used in the program must be declared in the declaration part. Thus, for each object, there is a parse subtree which contains its declaration. The ´reference to the declaration tree´ is the reference of the root of that parse subtree.

The level number is highly related with the way the syntax analyser works. The analyser create the symbol table sequentially, following the structure of the declaration part. Therefore, the order of the names in the symbol table is the image of the order of appearence of the names in the declaration part. This property of the symbol table will be used to implement the C-like record structure. Consedering a variable 'D' whose type is STRUCTURE, and which contains two field named 'E' and 'F'. If the level number of the entry which denotes the variable 'D' is i, the level number of the entries which denote 'E' and 'F' will be i+1. Besides, we assume that the level number of an entry which denotes a variable which is not contained in any structures, is 1. Finally, we assume that the level number of all the entries that do not denote a variable, is also 1.

Example.

```
                                                       level    name
                                                       _____    ____
VAR a : INTEGER;                                         1       a
    b : REAL;                                            1       b
    c : STRUCTURE                                        1       c
                ca : BOOLEAN;                            2       ca
                cb : STRUCTURE                           2       cb
                               cba : STRING;             3       cba
                               cbb : STRING;             3       cbb
                     END;
                cc : STRUCTURE                           2       cc
                               cca : BOOLEAN;            3       cca
                               ccb : INTEGER;            3       ccb
                     END;
        END;
    d : ARRAY [3,5] OF REAL;                             1       d
```

2.2. Access routines.

The routines that work on the symbol table are :

1. CREATE(str, leng, class, niv, ref) which allows one to create an entry in the symbol table.

   Inputs :

   - str : contains the string which represents the name.
   - leng : contains the LENGTH of 'str'.
   - class : indicates the CLASS of the name.
   - niv : contains the LEVEL number.
   - ref : contains the reference of the root of the declaration tree.

   Output :

   The output is the reference of the created entry.

2. FIND(ref, str, leng) which allows one to get the reference of the entry of a particular name.

   Inputs :

   - ref : contains the reference of the starting entry in the symbol table.
   - leng : contains the LENGTH of 'str'.
   - str : contains the string which represents the name to search out.

   Output :

   The output is the reference of the entry if the search was successful, otherwise the output is a 'NUL' value.

3. TREE(ref) which allows one to get the reference of the root of the declaration tree.

   Input :

   - ref : contains the reference of the entry in the symbol table.

   Output :

   The output is the reference of the root of the declaration subtree. If ref is invalid, the output is a NUL value.

4. CLASS(ref) which allows one to get the CLASS value of an entry.

   Input :

   - ref : contains the reference of the entry in the symbol table.

   Output :

   The output is the CLASS value of the specified entry. If ref is invalid, the output is a NUL value.

5. CTREE(ref,node) which allows one to update the field which contains the reference of the root of the declaration tree, for a particular entry in the symbol table.

   Inputs :

   - ref : contains the reference of the entry in the symbol table.

   - node : contains the reference of the root of the declaration subtree that defines the properties of objects depicted by the specified entry.

   Output :

   - 0 means that the reference ('ref') of the entry was valid.
   - 1 means that the reference ('ref') of the entry was invalid.

6. DEL(ref) which allows one to delete an entry in the symbol table.

   Inputs :

   - ref : contains the reference of the entry in the symbol table.

   Output :

   - 0 means that the reference ('ref') of the entry was valid.
   - 1 means that the reference ('ref') of the entry was invalid.

7. **NEXT(ref)** which allows one to get the following entry of a particular entry in the symbol table.

<u>Inputs</u> :

- ref : contains the reference of the entry in the symbol table.

<u>Output</u> :

- a 'NUL' value, if 'ref' is invalid,
- otherwise the output is the reference of the entry following the given entry.

3. THE LEXICAL ANALYSER.

The lexical analyser is the interface between the source program and the syntax analyser. The lexical analyser reads the source program one character at a time, carving the source program into a sequence of atomic units called tokens. Each token consists of a sequence of characters, and can be considered as a single logical entity. Names, keywords, constants, operators, and punctuation symbols such as commas and parenthesis are typical tokens. for example, in ADL statement

```
FOR cus:=customer DO
```

we find the following five tokens :

1) FOR
2) cus
3) :=
4) customer
5) DO

There are two kinds of token : specific strings such as IF or semicolon, and classes of strings such as names or constants. To handle both cases, we shall treat a token as a pair consisting of two parts : a token type, and a token value. For convenience, a token consisting of a specific string such as a semicolon will be treated as having a type but no value. A token such as the name ´cus´, above, has a type NAME and a value consisting of the string ´cus´. Frequently, we shall refer to the type or value as the token itself. Thus, when we talk about name being a token, we are referring to a token whose value is ´cus´.

An important part of the input process is carried out by the lexical analyser. It reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. There is a considerable leeway in deciding whether to recognize structures using the lexical analyser (regular expressions) or the syntax analyser (grammar rules). The complete set of lower level structures that we shall consider as token is given -using LEX constructs, in annexe [2].

the lexical analyser and the following phase, the syntax analyser, are often group together into the same pass. In that pass, the lexical analyser operates under the control of the parser. The parser asks the lexical analyser for the next token, whenever the parser needs one. The lexical analyser returns to the parser a code for the token that it found. In the case that the token is a name or another token with a value, the value is also returned to the parser.

4. SYNTAX ANALYSIS.

The syntax analyser has two functions. It checks that the tokens appearing in its input, which is the output of the lexical analyser, occur in patterns that match the specification of the source language. It also imposes on the tokens a tree structure that is used by subsequent phases.

For example, if an ADL program contains the expression

```
IF a > b THEN THEN
```

then, after lexical analysis, this expression might appear to the syntax analyser as the token sequence

```
IF name > name THEN THEN
```

on seeing the second THEN, the syntax analyser should detect an error situation, because the presence of these two adjacent THEN vioaltes the formation rules of a IF statement.

The second aspect of syntax analysis, is to make explicit the hierachical structure of the incoming token stream by identifying which part of the token stream should be group together. The language specification must tell us what hierachical structure each source program has. These rules form the syntactic specification of a programming language.

The parsing method that we develop is called shift-reduce parsing, because it attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). We can think of this process as one of "reducing" a string w to the start symbol of a grammar. At each step a string matching the right side of a production is replaced by the symbol on the left. Detailed informations about parsing techniques are given in [AHO.77].

This section discusses the type of "problems" we had to deal with, and how we solved them, using YACC. The complete description of the syntax analyser using YACC is given in annexe [3].

4.1. Constructing a parse tree.

The strategy is to maintain a 'forest ' of partially-completed derivation trees as we parse bottom-up. With each symbol on the stack we associate a pointer to a tree whose root is that symbol and whose yield is the string of terminals which have been reduced to that symbol, perhaps by a long series of reductions. At the end of the shift-reduce parse, the start symbol remaining on the stack will have the entire parse tree associated with it. The bottom-up tree construction process has two aspects.

1. When we shift an input symbol 'a' onto the stack we create a one-node tree labeled 'a'. Both the root and the yield of these tree are 'a', and the yield truly represents the string of terminals "reduced" (by zero reductions) to the symbol 'a'.

2. When we reduce X1 X2 X3 ... Xn to A, we create a new node labeled A. Its children, from left to right, are the roots of the trees for X1 X2 X3 ... Xn. If for all 'i' the tree for Xi has yield xi, then the yield for the new tree is x1 x2 x3 ... xn. This string has in fact been reduced to A by a series of reductions culmunating in the present one. As a special case, if we reduce {} to A we create a node labeled A with one child labeled {}.

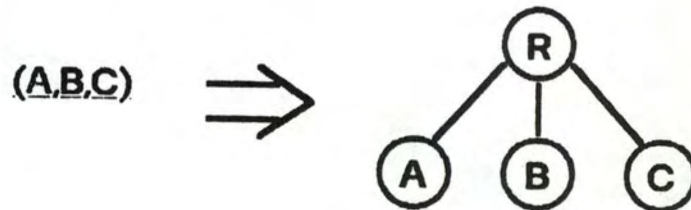
4.2. Recursive definition of lists.

The goal of the statements associated with the definition of a list, is to produce a ordinary tree which has the following properties :

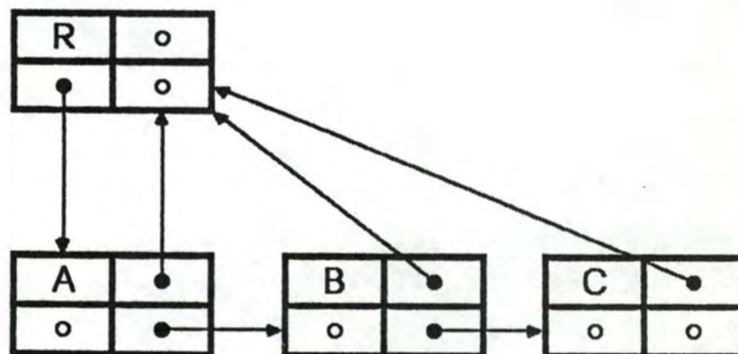
- the root, which indicates that the subtree is a list, is not a node that represents an element of the list. The root is especially dedicated to be the head of the list.

- the son of that root is a node that represents an element of the list.

Example :



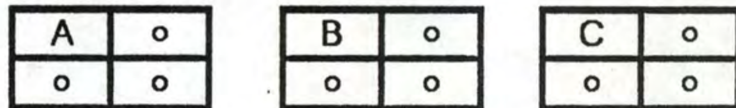
Since we used binary trees to implement ordinary trees, the internal representation will be,



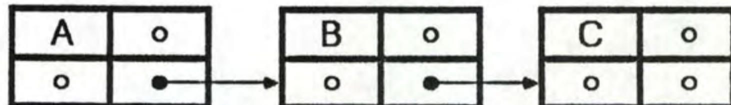
The creation of a list consists of three steps. Here they are, in chronological order :

- The creation of the subtrees that represent the elements of the list. To each element correspond one subtree.
- The linkage (by a 'BROTHER' link) of the roots of the subtrees that represents the elements of the list.
- The creation of the root of the list, and linkage (by 'SON' and 'FATHER' links) of that root to the roots of the list previously defined.

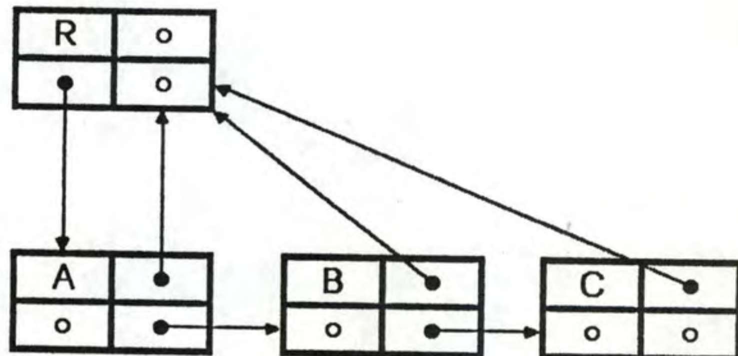
1. Creation of the elements.



2. Linkage of the elements.



3. Creation of the root.



Following this three step process, we used three grammar rules to define a list.

```
<list>     ::= <list_of_element>

              {- create  a root node whose type
                 is 'head of list', and link the
                 node to its sons. the first
                 element of the list is given by $1.    }


<list_of element> ::= <element> , <list_of_element>

              {-link the brothers $1,$3.

               -return $1.(which is the reference of
                the first node of <list_of_element>).  }

                  | <element>

              {-return $1.   }


<element>  ::=   <element_def>

              {-create a node which is the root of the
                subtree that represents the element.

               -return the reference of the node created.  }
```

4.3. managing names in declarations and in statements.

The symbol table is filled while parsing the declaration part. For each name encoutereed, an entry is created in the symbol table. While parsing the statement part, the symbol table is used to check if each name encountereed was previously declared. Besides, the node that represents the name in the parse tree, will be updated so that the field 'SON' will point to the entry -in the symbol table, that describes the name.

4.4. managing record structures in declaration and in statements.

4.4.1. In the declaration part.

Since the entries in the symbol table are created in the order of appearence of the corresponding names, the only thing that might had cause problems in managing record structure, was the level number. We had to found a mechanism which was able to give the level number of a name when creating the corresponding entry. The mechanism we implemented is simple. Each time the parser encounts the token STRUCTURE (which denotes the begining of a structure), it will increment a global variable -called level_number, by one. Each time the parser encounts a token END that denotes the end of a structure, it will decrements level_number by one. Thus, we are sure that we know, at any given time, what is the level we are working at.

4.4.2. In statement part.

In the statement part, the token that denotes a jump to a lower level, is a period between two names. The form a.b denotes the field ´b´, of the structured variable ´a´.Thus, the name ´b´ has to be searched out in the portion of names that are just after the ´a´ -in the symbol table. Moreover, the level number of the entry that is searched out must be equal to one plus the level number of ´a´. That´s why the routine FIND (whose job is to search out the entry that correspond to a given name) always asks for two parameters :

- The name to be search out.
- The reference of the entry from which the search has to start.

The routine will start from the reference of the entry indicated by the second parameter. It will search out an entry that i)contains a name corresponding to the first parameter,ii)contains a level number equal to one plus the level number of the entry specified by the second parameter. It will stop if it finds a entry that matches the specification above (successful search), or if it encounts an entry that contains the same level number that the starting entry (unsuccessful search), or, obviously, if it reaches the end of the symbol table.

Remark.

To find a name that has the level number one, one will just put a ´NUL´ value in the second parameter.

4.5. Managing constant.

The data structure where constant values are stored, is called the constant table. For each constant encountereed, an entry is created in the constant table. The reference of the entry is stored in the field 'SON' of the node which represents the constant. In order to be able to use the constant table, we need the two following routines :

- PUTK(txt,leng) which allows one to add a constant to the constant table.

  Inputs

  -txt : array of characters that denote the constant.

  -leng : integer variable which contains the length of the constant.

  Outputs :

  -the reference of the created entry in the constant table.

- GETK(ref) which allows one to get the constant that correspond to the given reference.

  Inputs :
  -ref : contains the reference of the created entry in the constant table.

  Ouputs :

  The output is a record that contains :
  -txt : array of characters that denote the constant.

  -leng : integer variable which contains the length of the constant.

5. SEMANTIC ANALYSIS.

The term semantic analysis is applied to the determination of the type of intermediate results, the check that arguments are of type that are legal for an application of an operator, and determination of the operation denoted by the operator . Semantic analysis can be done during the syntax analysis phase or after.

Since the semantic analysis is still in a design phase, we do not have an implementation to present.

Chapter 10: THE LOGICAL RECORD ACCESS EVALUATOR

1. Introduction

The first section of this chapter explains the goal of the evaluator. In the second section, we explain the problem of the evaluation by mean of an example. An evaluation method is presented in the next section. Finally, we criticize the proposed evaluation method.

2. Goal of the evaluation

Before giving the goal of the evaluation, we will define the concepts of "logical record access (LRA)" as opposed to "physical record access", and the relationship between them.

A logical record access (LRA see [TEO.83]) is an access to a record of the database as it is perceived by an application program, programmer, or user. One logical record access may generate several physical record accesses due to physical access techniques (index, intermediate record types,...).
For us, a logical record access will be an access to a record type of a GAM database schema (see chapter 2) using the access methods offered by the GAM model : sequential access, access by access key, access by access path. We are not concerned by the number of physical record accesses that could be performed by physical access techniques which implement these methods.

The relationship between logical record accesses and physical I/O depends on many factors, such as block size and physical clustering. These factors are unknown at the logical design layer.

The goal of the evaluation is to provide the database designer with a criteria which enable him to choose among several implementation corresponding to the same set of process specification. One implementation consists of a data structure and a set of algorithms.
The criteria is the minimum number of logical record accesses to the database.

Although it is not absolute, there is a good probability that an implementation minimizing the number of LRA, minimizes also the number of physical accesses to the database.

3. Example and problem

In this section we first reformulate the goal of the evaluation in terms of the logical database design methodology developed in Namur. Then, we explain the basics of the evaluation process. Finally we give an example of evaluation showing the line of arguments which led us to a stepwize evaluation method in parallel with the transformation of high level algorithms to low level algorithms.

3.1. Reformulated goal of the evaluation

As it is presented in chapter 5, the goal of the evaluation is to select the best element of a set of pairs (effective algorithm, necessary accesses schema (NAS)) corresponding to one pair (predicative algorithm, possible accesses schema (PAS)). The selection criteria is the minimum number of logical record access (LRA) to the database, during one execution of the effective algorithm.

The basic information needed for the evaluation consists of the statistical description of the database schema (see chapter 4), and the tree representation of the effective algorithm (output of the ADL analyser presented in chapter 9).

An effective algorithm is the result of the transformation process of a predicative algorithm. The transformation process is the implementation of the access strategies choosed by the database designer to implement the predicates.

3.2. Basics of the evaluation

Theoretically, the evaluation of ADL algorithms can be reduced to the evaluation of three algorithm structures : the sequence, the loop, and the alternative. We will give an abstract of the method used to evaluate these structures. We hope, it will make the reader familiar with the concept of evaluation. Only the basics are given. The evaluation performed by the evaluator is more sophisticated (see [HAI.76A], and [KER.83]).

- Evaluation of a sequence of ADL statements :

  The expected number of LRA performed during one execution of the sequence is the sum of the expected numbers of LRA performed by each statements of the sequence.

- <u>Evaluation of an IF statement</u> :

  Given the following IF statement :

  if <CONDITION> then <SEQUENCE-1>
  else <SEQUENCE-2>

  <CONDITION> is the condition which controls the IF statement.

  <SEQUENCE-1> is the sequence of statements executed if the condition has the value true.

  <SEQUENCE-2> is the sequence of statements executed if the condition has the value false.

  N, the expected number of LRA performed during one execution of the statement, depends on :

  P = probability that the condition has the value true.

  NS1 = expected number of LRA performed during one execution of <SEQUENCE-1>.

  NS2 = expected number of LRA performed during one execution of <SEQUENCE-2>.

  C = expected number of LRA performed in order to check if the condition is true. The fact that a condition has a cost in terms of LRA is due to the power of ADL. The language enable the expression of conditions on database objects (see chapter 3).

  N is given by :

  N = C + P*NS1 + (1-P)*NS2

- **Evaluation of a FOR statement :**

  Given the following FOR statement :

  for V := <COLLECTION> do

  <SEQUENCE>

  endfor V

  V is the loop variable.

  <COLLECTION> is the designation of the collection of objects which controls the loop.

  <SEQUENCE> is the sequence of statements controlled by the loop.

  N, the expected number of LRA performed during one execution of the FOR statement, depends on :

  NS = expected number of LRA performed during one execution of <SEQUENCE>.

  SC = expected size of the collection.

  C = expected number of LRA performed in order to build the collection. This cost in terms of LRA is due to the fact that ADL allows loop statements driven by collections of database objects (see chapter 3).

  N is given by :

  N = C + NS * SC

3.3. Example :

Remark : for simplicity, only the statement part of the algorithms is given.

Possible accesses schema

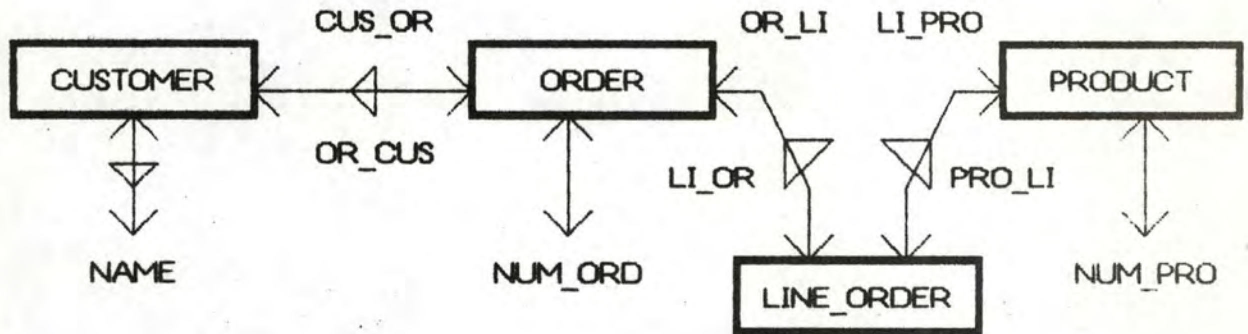


fig 10.1

Predicative algorithm

```
for CUS := CUSTOMER(CUS_OR: ORDER(OR_LI: LINE_ORDER(LI_PRO:
                                PRODUCT(:NUM_PRO = 783 )))) do

        print (NAME(:CUS))

endfor CUS
```

The purpose of this algorithm is to print the names of the customers who have ordered the product identified by the number of product 783. The evaluation of the expected number of LRA performed by one execution of this algorithm is impossible. The sequences of access to the different record types of the PAS is not specified.

Necessary accesses schema

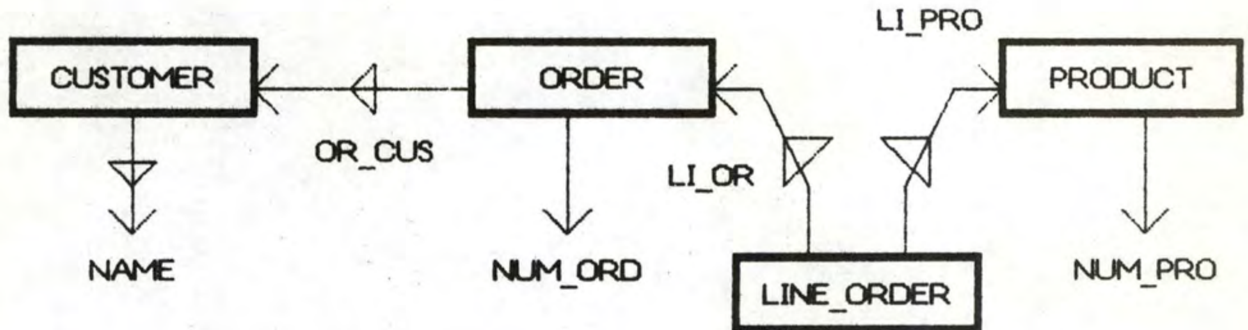


fig 10.2

Effective algorithm

```
NEWLIST(CUSTOMER,LIST_CUSTOMER)
for LINE := LINE_ORDER do
   LINE_OK := false

   for PRO := PRODUCT(PRO_LI :LINE) do
      if PRO( :NUM_PRO = 783) then
         LINE_OK := true
         exit PRO
      endif
   endfor PRO

   if LINE_OK then

      for O := ORDER(OR_LI: LINE) do
         for CUS := CUSTOMER(CUS_OR: O) do
            if  CUS not in LIST_CUSTOMER then
               ADDLIST(CUS,LIST_CUSTOMER)
               print(NAME( :CUS))
            endif
         endfor CUS
      endfor O

   endif

endfor LINE
```

We will first explain the effective algorithm in a non formal way. Later, we determine what can be or cannot be evaluated.

The effective algorithm correspond to the following sequence of actions :

- a sequential access to all the LINE_ORDER records contained in the database (LINE loop).
- for each LINE_ORDER record, access is made to the related PRODUCT records in order to check if the number of the product is 783 (PRO loop).
- if the product number is 783, the execution of the PRO loop is ended. The algorithm access to the CUSTOMER records related to the LINE_ORDER record via LI_OR and OR_CUS (O loop and CUS loop).
- a last check is made to determine if the CUSTOMER does not belong to the list of customers for which the name has already been printed (if CUS not in LIST_CUSTOMER). As a matter of fact, a customer may have ordered the same product several times.

Here comes the list of the effective algorithm parts that can be evaluated if no account is taken of the correlations.

- the number of executions of the LINE loop : it is the number of LINE_ORDER records in the database. A value of this number is given in the statistical description of the database schema.
- the number of executions of the PRO loop. This number depends on the expected number of PRODUCT records related to one LINE_ORDER record and on the probability that the number of a product has the value 783. The first value derives from the distribution function describing the access path type LI_PRO. The second value can be obtained from the distribution of the values of the NUM_PRO data item.
- In the same way, the numbers of executions of the O loop and CUS loop are easy to compute from the statistical description.

Because no distribution function of the values of the switch LINE_OK is given in the statistical description, we cannot estimate the probability that the switch has the value true. The same problem arise for the number of customer names printed by the print(NAME(:CUS)) statement. We lack of statistical information for the list LIST_CUSTOMER.

Obviously, we know that the number of customer names printed is equal to the number of customer qualifying the only predicate of the predicative algorithm. An estimation of the expected value of this number can be derived from the statistical description. Trying to estimate this number giving only the tree representation of the effective algorithm could lead to erroneous values or implies some kind of pattern recognition.
The probability of the switch LINE_OK is also easy to obtain if we note that it is the probability that a line qualify the predicate :

LINE(LI_PRO: PRODUCT(:NUM_PRO = 783))

These examples show that it is easier to compute the probability of a condition starting from its explicit expression rather than from its algorithmic development.

The two above evaluation problems are side effects of the transformation process from predicative algorithm into effective algorithm. These impedements for an automatic evaluator of effective algorithms are overcomed by the database designer because of its global view of the transformation/evaluation process.

We are faced with a dilemma. Generally, an evaluator cannot produce results if it takes as only input for the description of the algorithm, either the predicative algorithm, or the effective algorithm. One solution is to make the evaluation in parallel with the transformation process. We have called this solution "The stepwize evaluation method". A presentation of the method is given in the next section.

4. The evaluation method

The example of section 3 shows that the evaluation of ADL statements is frequently impossible. This is due rather to the fact that we lack of statistical information about the objects manipulated by the statements, than to the complexity of the evaluation itself.

The evaluation of effective algorithms can be divided into two smaller problems :

- First, the evaluation of ADL expressions and statements. This problem has been resolved by professor HAINAUT [HAI.76A] and developed by Anne KERSTENNE [KER.83]. The result of these works is a set of formulas which estimate the expected number of LRA performed by the execution of each ADL statement or expression, and a statistical package designed to compute the formulas.

- Evaluation of a whole algorithm is the second problem. Although access strategies defined in the effective form of the algorithm are neccessary for the evaluation; information valuable for the evaluator is given by the predicative algorithm. This fact leaded us to a stepwize evaluation method starting from the partial evaluation of the predicative algorithm and ending by the evaluation of the correponding effective algorithm.

In this section, we give :

- a summary of the formulas computed by the statistical package.
- the stepwize evaluation method.

4.1. Formulas computed by the statistical package.

The statistical package performs calculations of probability and cost, in terms of LRA, of ADL statements. The package was developed by Anne KERSTENNE. Its complete description is to be founded in [KER.83].

For each calculation, we will give :

- The syntax of the concern statement
- The semantic
- One example
- The call to be performed with the description of the parameters.

The package is supposed to perform all the computations we need to evaluate access algorithms.

Reading of chapters 2, 3 and 4 is a prerequisite for the understanding of this section.

4.1.1. Probability of a cardinal relation criteria

Syntax : (R: <K> B(CB))

R : relationship from A -> B ID-JD, II-JI
<K> : is a cardinal i-j (i,j integer)
0 <= i <= j <= JD
CB : is a condition on the objects designated by B

Semantic : The criteria is true for one occurence of A if it is related to n occurences of B which satisfy CB condition. The number n is an element of [i:j].

e.g. : CUSTOMER(CO: [5:*] (:NUM-ORDER > 100))

<=> customers who have five or more orders with order numbers higher than 100.

Call : The probability of the cardinal relation criteria is computed by the PROBCD procedure of the statistical package.

PROBCD (IDENT,P,CAR,I,J,RES)

Input :

- IDENT is a one-dimensional array :

  IDENT(1) = database key of the "statistic description" record corresponding to the R relationship.
  IDENT(2) = 0 if access path type description
  1 if item description
  IDENT(3) = 1 if direct relation
  -1 if inverse relation

- P = probability of CB condition.

- CAR = true if cardinal
  = false if ordinal

- I,J are the lower and upper limits of <K> cardinal

Output :

- RES : gives the probability of the criteria.

4.1.2. Probability of an ordinal relation criteria

Syntax : (R: <O> B(CB))

R : relationship from A -> B ID-JD, II-JI
<O> : is an ordinal i-j (i,j integer 0
0 <= i <= j <= JD
CB : is a condition on B objects

Semantic :

The criteria is true for one occurence of A if :

- it is related to k occurences of B
- i <= k
- the occurences of B related with A which have a rank between i and inf(k,j) satisfies the Cb condition.

e.g. : CUSTOMER(CO: #[1-5] ORDER(:TOTAL-COST < 1000))

<=> Customer who have their first five orders with a total cost lower than 1000.

Call : The probability of the ordinal relation criteria is computed by the PROBCD procedure of the statistical package.

PROBCD (IDENT,P,CAR,I,J,RES)

The definition of the parameters are the same as in (3.1.1).

4.1.3. Probability of a belonging criteria

Syntax : OP E

OP : Operator

- only "in" or "not in" for complex objects
- =, <, <=, >, >=, =/=, in [a,b] for item values.

E : Collection of objects.
For simplicity, we use only collections containing only one object or ranges for item values.

Semantic : The criteria is true if the object belongs (or do not belong) to the collection of objects defined by the criteria.

e.g. : STUDENT(: CITY = "ANN ARBOR")

<=> Students living in Ann Arbor.

Call : The probability of the belonging criteria is computed by the PROBAP procedure of the statistical package.

PROBAP (IDENT,OP,TA,TB,N,RES)

Input :

-IDENT : same as above.

- OP : gives the type of the operator.

```
OP = 1  <->  =  or "in"
OP = 2  <->  =/=  or "not in"
OP = 3  <->  <
OP = 4  <->  <=
OP = 5  <->  >
OP = 6  <->  >=
OP = 7  <->  in [a,b]
```

- TA and TB are integer arrays used for the storage of "a" value and "b" value for integer, real, or string item values.

- N gives the number of useful values in TA and TB arrays.

Output :

- RES is the probability of the criteria.

If ident(2) = 0

we are checking if a complex object is equal or not to another complex object.

The probability for equality is :

1/(number of objects of that type in the database)

So, we do not use TA, TB, and N.

If ident(2) = 1

We are checking if an elementary object belongs to a range of values.

TA and TB must always represent a value of the same type as the data item. TB has a value only if OP = 7 (to specify a range without implicit lower or higher bounds).

If the data item is alphanumeric with length k, then N=k and for each i (1 <= i <= k), TA(i) = i(th) character of the alphanumeric string.

If the data item is an integer, then N=1 and TA(1) = a.

If the data item is real, then N=2 and TA(1) * 10**TA(2).

4.1.4. Cost of an access loop

Syntax :

```
for VAR := <*/O> B((R:RECVAR) and (CB))
     S
endfor VAR
```

R : relationship from A -> B.
* : is the ALL cardinal
O : is an ordinal i,j
CB : is a condition on B objects
RECVAR : is a record variable containing an A object
VAR : is the loop variable
S : is a sequence of ADL statements

Semantic : Execute the S sequence for each occurence of B, or for occurences of B with rank between i and j, accessed by R from RECVAR and satisfying the CB condition. The object type A must be a record type.

e.g. :

```
CUS := CUSTOMER (:NAME = "SMITH")
for O := ORDER ((CO:CUS) and (:TOTAL-COST > 100))
     S
endfor O
```

<=> execute the S sequence for all the orders of customer SMITH if their total-cost is greater than 100.

Call : The cost of the access loop is computed by the BOUCLE procedure if S doesn't contain next or exit statements.

BOUCLE (IDENT,P,CAR,I,J,COMPT,RES)

Input :

- IDENT, P, CAR, I, J are the same as above.
- COMPT is true if there is a target counter in A.

Output :

- RES(1) = m' : mean number of B selected => mean number of execution of the loop.
- RES(2) = m" : meam number of access to do for one origin of R.

The cost of an access loop with next and exit statements is computed by the NEXTEX procedure.

NEXTEX (IDENT,P,CAR,I,J,COMPT,PN,RES)

- **IDENT,P,CAR,I,J,COMPT,RES are the same as above.**
- **PN : is the probability that one execution of the sequence ends by a next statement.**

4.1.5. Number of distinct elements in a collection

Syntax :

```
for VAR := <*/O> B((R:A(CA) and (CB))
    S
endfor VAR
```

R : is a relationship from A -> B
CA : is a condition on A objects
CB : is a condition on B objects
VAR : is the loop variable
* : is the ALL cardinal
O : is an ordinal i,j

Semantic : The collection is the collection of B objects selected by the loop. Namely, the B objects related by R to at least one A of the A(CA) collection, satisfying the CB condition and with rank between i and j.

e.g.

```
for C:=#5:10 CUSTOMER((CO:ORDER(:NUM_ORDER > 1000)
                                and (:CITY="DETROIT"))
    S
endfor C
```

<=> the fifth to the tenth CUSTOMER related to an order with order number greater than 1000 and living in Detroit.

Call : The number of distinct elements in the collection is computed by the ELLCOLL procedure.

ELLCOLL (IDENT,P,CAR,I,J,COMPT,PN,PO,RES)

Input :

- IDENT,CAR,I,J,COMPT are the same as above.
- PO : is the probability of CA condition.
- PN : is the probability of a next statement.
- P : is the probability of CB condition.

Output :

- RES : is the proportion of B objects selected by the loop. Namely, the number of B objects selected by the loop divided by the total number of B objects in the database.

4.1.6. Cost of the evaluation of a condition

The cost of evaluation of a belonging criteria is supposed to be zero (computation in central memory). We will compute only the cost of a relation criteria.

Syntax : (R: <K/O> B(CB))

see sections 4.1.1 and 4.1.2

Semantic : see sections 4.1.1 and 4.1.2

e.g. : see sections 4.1.1 and 4.1.2

Call : The cost of evaluation of a cardinal or ordinal relation criteria is computed by the COUTCD procedure of the statistical package.

COUTCD (IDENT,P,CAR,I,J,COMPT,RES)

Input :

- P, IDENT, CAR, I, J, COMPT, are the same as above.

Output :

- RES(1) = m1 : the expected number of targets to be accessed via R to evaluate the criteria.
- RES(2) = m2 : the expected number of targets for which CB will be evaluated.

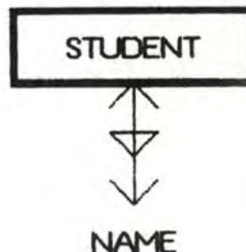
4.2. The stepwize evaluation method

First of all, we will more precesely define some concepts already used in section 3, and some new concepts. Then, we will present the steps of the evaluation method.

4.2.1. Concepts definition

- Access strategies : An access strategy specifies the sequence and the type of accesses to the records of a database. It specifies a way to build a collection of records, or check a condition.

  Example :



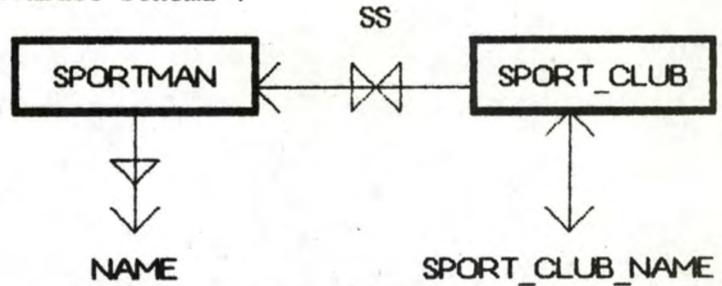
  Given the above simple database schema, we can imagine two access strategies to obtain the collection of students with NAME = "JACKSON". The first access strategy would be to access sequentially to all the records STUDENT. For each accessed record, check if its NAME data item has the specified value. The second access strategy would be to use the access key mechanism in order to access to each record STUDENT with NAME = "JACKSON". One can imagine the difference in terms of LRA between these two implementations.

- The transformation process : It is the process that takes as input a predicative algorithm, a possible accesses schema (PAS), the specification of the access strategies desired to implement the predicates and produce as output an effective algorithm and a necessary accesses schema (NAS). The effective algorithm is the implementation of predicative algorithm according to the access strategies. The NAS is the restriction of the PAS to the access structures needed by the effective algorithm.

- One transformation : Denotes the transformation of one part of an algorithm according to one access strategy.

  Example :

  - database schema :



  - part of the algorithm before transformation :

```
<SEQUENCE 1>

for SM := SPORTMAN(SS: SPORT_CLUB(:
          SPORT_CLUB_NAME = "SANTA MONICA")) do

   <SEQUENCE 2>

endfor SM

<SEQUENCE 3>
```

  - access strategy :

    First, access to the SPORT_CLUB records by the SPORT_CLUB_NAME access key. Then, access to the SPORTMAN records by the access path SS.

- equivalent part of the algorithm after transformation :

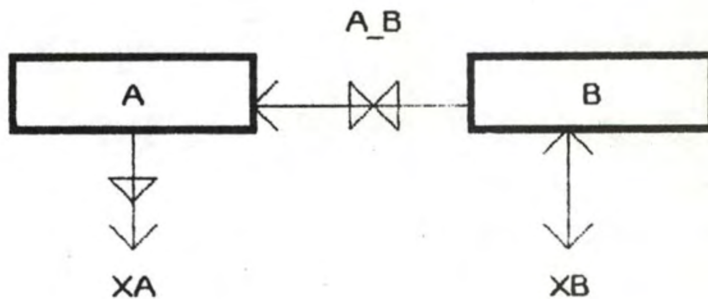
```
<SEQUENCE 1>

for SC := (:SPORT_CLUB_NAME = "SANTA MONICA") do

   for SM := SPORTMAN(SS: SC) do

      <SEQUENCE 2>

   endfor SM

endfor SC

<SEQUENCE 3>
```

- __Type of transformation__ : A type of transformation defines the common properties of a set of transformations.

  __Example__ :

  The above example of transformation belongs to the following transformation type :

  - database schema :



- part of the algorithm before transformation :

```
<SEQUENCE 1>

for V := A(A-B: B(:XB = X)) do

   <SEQUENCE 2>

endfor V

<SEQUENCE 3>
```

- access strategy :

  First, access to the B record by the XB access key. Then, access to the A records by the access path A-B.

- equivalent part of the algorithm after transformation :

```
<SEQUENCE 1>
for VB := B(:XB = X) do

   for VA := A(A-B: VB) do

      <SEQUENCE 2>

   endfor VA

endfor VB

<SEQUENCE 3>
```

4.2.2. The steps of the evaluation method

The stepwize evaluation method is composed of the following steps :

- i) Partial evaluation of the predicative algorithm.
- ii) Transformation process from predicative to effective algorithm.
- iii) Evaluation of the effective algorithm.

- i) **Partial evaluation of the predicative algorithm**

During this first step, we evaluate the predicative algorithm structures which can be already evaluated. This sets the problem of "what can be evaluated".

Of course, we can only estimate values which depend on the data properties described in the statistical description of the database schema.

It is possible, for each algorithm level, to estimate the size of a collection or the probability of a condition, if these values depend on data properties statistically described. But the cost of building a collection or checking a condition is not evaluable unless the access strategy is known. This implies that LRA counts could not be completed before the access strategies are made explicit by the effective algorithm.

Probability of conditions depending on the logic of the application and not on the properties of the database contents are, as we said before, very difficult to evaluate (ex. the LINE-OK switch of section 3). Still, we will be able to estimate most of these probabilities.

We consider two kinds of conditions apparently independent of the database contents properties :

- i) condition which are part of the result of a known transformation of an ADL piece of code.

  Ex : suppose that the piece of ADL code :

```
for VC := C(RDC: D(CD)) do

   S1

endfor VC
```

  is transformed in the following algorithm :

```
for VC := C do

   C-OK := false

   for VD := D(RDC: VC) do

      if VD(CD) then C-OK := true
                     exit VD
                endif
   endfor VD

   if C-OK then

            S1

         endif

endfor VC
```

The desired collection of C objects is obtained by accessing sequentially to all the C objects and then checking if the current C is related to a D object which qualify the CD condition.

If the evaluator is aware that the second piece of code is the result of a known transformation of the first one; the estimation of the probability that C-OK has the value true is easy to obtain. No unknown information was added by the transformation.

The concept of known transformation is essential for the evaluation. It is the only way to associate the values estimated during the partial evaluation to the ADL constructs generated by the transformation process.

- ii) condition independent of the database contents properties which are not the result of a known transformation. These conditions are either the result of an unknown transformation or they are part of the predicative algorithm. Their probability cannot be automatically evaluated. An estimation of their probability will be asked to the database designer.

Ex : given the database schema of figure 10.1, we can imagine the following predicative algorithm :

```
for O := ORDER do

   if NAME(:CUSTOMER(CUS-OR:O)) = "SMITH" then
      O-OK := true
   endif

endfor O

if O-OK then print (NUM-ORD(:O))
endif
```

We suppose that the second type of condition will be rarely found in database access algorithms.

The input of the partial evaluation consists of :

- the tree representation of the predicative algorithm.
- the possible accesses schema (PAS).
- the statistical description of the PAS.

The output consists of :

- the values which have been estimated. These values are associated to the corresponding statements and expressions of the predicative algorithm.

- ii) Transformation process

This process takes as input :

- the tree representation of the predicative algorithm.
- the values already evaluated.
- the specifications of the access strategies.

It produces as result :

- the tree representation of the generated effective algorithm
- the values already evaluated during the first step. These values are associated to the corresponding statements and expressions of the effective algorithm.

As stated before, the transformation process must use known transformations. We will define more precisely this concept :

A known transformation belongs to a knowm transformation type.

A type of transformation is known if :

- the transformation rules have been defined and recorded.
- the transformation can be applied by a program.
- the results of the partial evaluation of the code before transformation can be associated, by the program, to the generated code.

The transformation process described in this section does not ensure that the generated effective algorithm is automatically evaluable.

The strong point of the process is the fact that the generated effective algorithm is automatically evaluable if the predicative algorithm does not contain conditions independent from the database contents properties.

- ***iii) Evaluation of the effective algorithm***

This process evaluates the expected number of LRA performed during one execution of the algorithm.

It takes as input :

- the tree representation of the effective algorithm
- the values already evaluated. These values are associated to the corresponding statements and expressions of the effective algorithm.
- the possible accesses schema (PAS).
- the statistical description of the PAS.

The outputs consists of :

- the expected number of LRA performed during one execution of the algorithm. This number may be divided into numbers of LRA performed to each record type of the database. It could also be interesting to value separately, the expected number of LRA, by type of access (read, modify, delete...).

All the evaluation problems resulting from the transformation process where resolved before this step of the method.

4.2.3. Implementation of the method

This section gives a proposal for the implementation of the stepwize evaluation method.

The method implies three problems :

- the transformation process from the predicative algorithm to the correponding effective algorithm has to be supervised. This will be done by what we call the "transformater".

- we have to determine the set of known transformation types. Because it depends on the application type, on the database designer and on the target DBMS, we suggest the use of a dynamic set of transformation types that could be upgraded or modified by the database designer.

- We need a way to identify the transformation types. This should be achieved by a well designed access strategy notation.

4.2.3.1. **Architecture of a tool implementing the stepwize evaluation method.**

First, we will give a schema of the architecture. Then, we will describe its components.

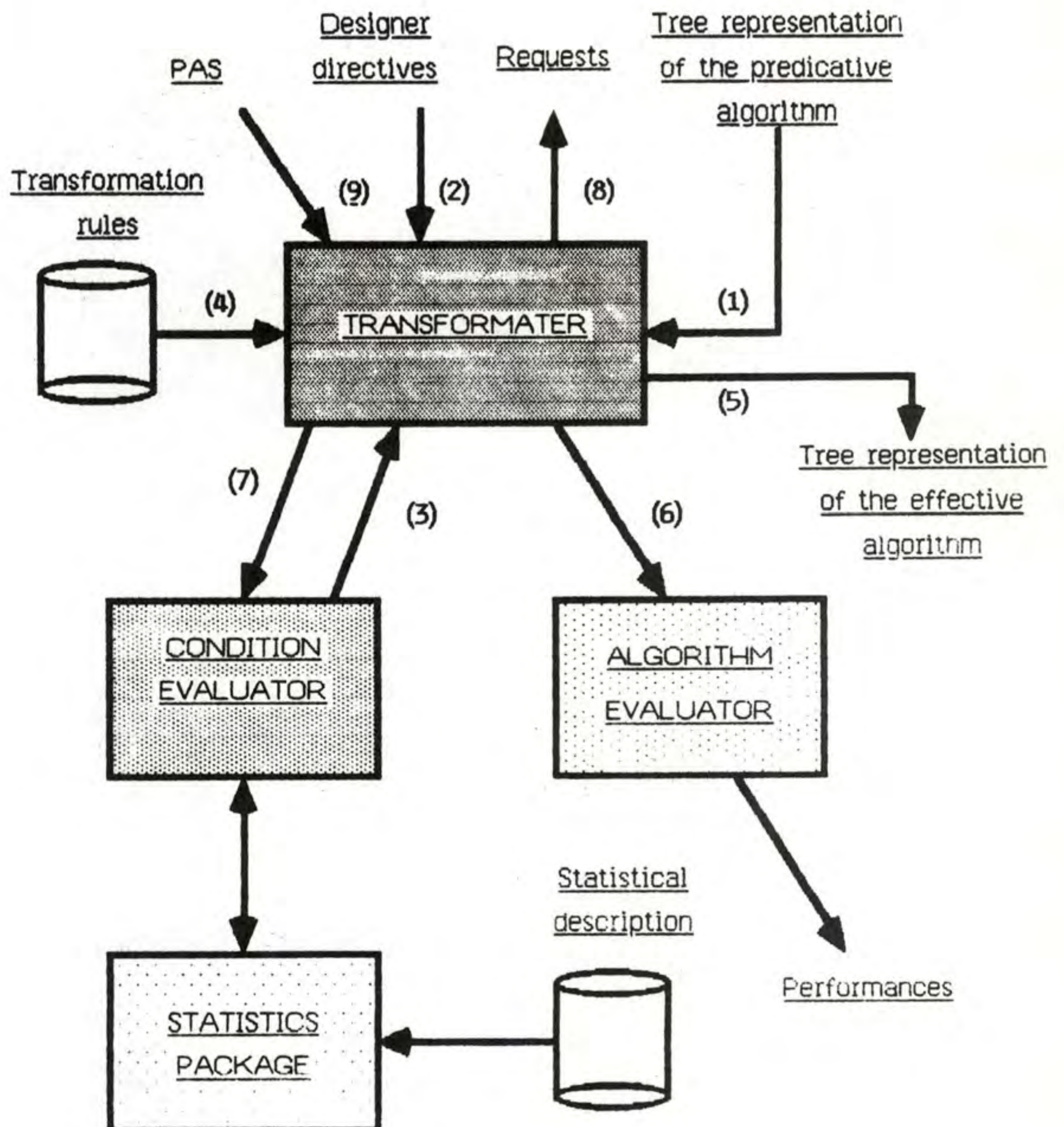


fig 10.3

Components :

- the statistics package computes the formulas presented in section 4.1.
- the condition evaluator evaluates the probabilities of arbitrary complex ADL conditions.
- the algorithm evaluator evaluates the effective algorithm.
- the transformater is the main part of the tool. It transforms a predicative algorithm into an effective algorithm. The transformater has two functions :
  - the transformation of a predicative algorithm into an effective algorithm.
  - keep track of valuable information, for the evaluator, which is lost after the execution of the transformations.

  The input of the transformater is composed by :

  - the tree representation of the predicative algorithm (1).
  - the possible accesses schema (9).
  - the directives of the database designer (2) consisting of :
    - the specifications of the access strategies.
    - the expected values of the probabilities which cannot be estimated by the evaluator.
  - the result of the evaluation of ADL expressions (3).
  - the transformation rules corresponding to the known transformation types (4).

  The ouput of the transformater is composed by :

  - the tree representation of the effective algorithm (5).
  - the result of the partial evaluation of the effective algorithm (6).

- the representations of ADL expressions that have to be evaluated (7).
- request for more information if an ADL expression cannot be evaluated (8).

5. Criticism

Evaluation of algorithms leads to two kinds of problems :

- the accuracy of the results.
- the feasibility of the evaluator.

We will develop these aspects.

5.1. Accuracy of the results

The accuracy of the results of the evaluation (namely, the estimated number of LRA to the database performed during one execution of an algorithm), is essentially limited by the size of the set of statistical information that can be reasonably asked to the database designer. The precision of the estimation depends on the statistical model used to describe the database contents (see chapter 4), and on the formulas computed by the statistical package (see section 4.1).

Of course, the size of the statistical description is proportional to its precision. We will distinguish tree levels of precision depending on on the mathematical tools used :

- mean values. This is the less precise description. It leads to erroneous evaluation [CHR.81].
- distribution functions. This is the basis of the statistical model presented in this thesis. We believe it is the optimal compromise between precision and size of the statistical description.
- multivariate distribution functions. These functions allow the expression of correlations between data item values [CHR.81].

The strong points of the statistical model presented in chapter four are :

- the sophisticated set of distribution functions used in order to describe the relation between two record types via an access path type, or the relation between a record type and its data items.
- the distribution of data item values, which is given by means of histograms, can be as precise as the database designer want it to be.

The drawbacks are :

- the statistical model concerns only the database contents, as such, it does not include the distribution of input values.

- the expression of correlations between data item values is not allowed by the statistical model.

The first of the above drawbacks is overcomed, in our evaluation method, by giving explicitly the probability of conditions involving input parameters.

A usefull improvement of the method should be, to add the distribution of the input values, along with the declaration of the input parameters in the declaration part of the algorithm [STA.83].

The second drawback is also overcomed by giving explicitly the probability of conditions which would lead to erroneous estimations if no account was taken of the correlations.

Two attempts to solve this problem has been given :

- Staniszkis proposes to allow, the database designer, to express the correlations between some data item values [STA.83]. We do not think it gives a better solution in the sense that the choice of the correlations that will be described implies an analysis of the algorithm. The complexity of such analysis is equivalent to the complexity of the detection of erroneous estimations in our method.

- Multivariate distributions have been used to describe the correlation between data items belonging to the same record type [CHR.81]. The desadvantage of this method are the amount of statistical information needed, and the complexity of the computations.

We think it is generally impossible to record all the correlations between the database objects. This fact is one of the arguments that led us to the conclusion that an evaluator should be a support for the evaluation rather than an automatic tool.

5.2. Feasability of the evaluator.

As it is shown in section 4.2.2, the estimation of the probability of conditions, which do not depend on the properties of the database contents, cannot be automated. Furthermore, the theory of Calculability demonstrates that an automatic procedure, which evaluate the performances of any algorithm, cannot be build.

The two above statements implies that the evaluation of ADL algorithms is a non structured problem that could not be solved by an automatic tool.

Conclusions.

Generally speaking, a database design process can be viewed as a series of steps, where one alternative among many is choosen. Therefore, at each step the designer has to i) enumerate alternatives, ii) evaluate these alternatives, and iii) choose one alternative.

The purpose of the documentation tool proposed in the first part of this work, is, to enable the designer to record alternatives by means of models. The GAM is dedicated to the description of database structures, while ADL supports the description of the database access algorithms.

Once recorded in a formal way, the alternatives can be processed in various manners, depending on i) the power of the models provided, and ii) the reports producers associated. Among these reports producers, one can find :

- queries, which are used to produce reports that describe the alternatives.

- analysers, which produce evaluations of the alternatives.

- generators, which produce new alternatives.

We implemented the documentation tool with a LSS generator, namely ISLDM/SEM (ISDOS MICHIGAN). The reports producers associated are only queries.

In the second part of the work, we have discussed the issues of our research on a particular analyser, namely, an Logical Record Access Evaluator. The first step consisted in defining an algoritm representation on which the evaluator could work. It appeared that a parse tree structure was appropriate. The ADL parser was achieve by means of a parser generator (YACC & LEX). We have showed some problems related to the evaluation of algorithms and we have proposed a stepwize evaluation method in parallel with the transformation from high level to low level algorithms. Although the method we presented solves the majority of the problems, we stated that the evaluation of arbitrary complex algorithms cannot be automated. This fact has strengthned our conviction that an LRA evaluator should be a support for the evaluation rather than an automatic tool. Furthermore, the concept of "known transformation" which appeared essential for the evaluation, has laid the basis of the automatic generation of low level algorithm.

Nevertheless, we believe that the database designer will still need to make many decisions and will need to adapt general principles to a specific environment. The skill of the designer will still be crucial.

[AHO.77] A.V. Aho and J.D. Ullman, Principles of compiler design, reading, MASS. : ADDISON-WESLEY 1977

[BOD.83] F. Bodart and Y. Pigneur,"Conception assistee des applications informatiques"
MASSON,Paris,1983.

[CHEN.76] P.P. Chen,"The Entity-Relationship Model"
ACM TODS, Vol.1, nr.1,1976.

[CHR.81] S. Christodoulakis,"Estimating selectivities in data bases"
PhD thesis, University of Toronto, 1981.

[HAI.84A] J-L. Hainaut,"Description du modele d'acces generalise" Lecture notes, FUNDP Namur.

[HAI.80A] J-L. Hainaut,"Derivation d'une premiere structure d'access a partir du modele Entite/Association"
Lecture notes, FUNDP Namur.

[HAI.76A] J-L. Hainaut,"Calculs des probabilites et des couts"
Internal paper, FUNDP Namur.

[JOHNS.75] S.C. Johnson,"YACC -yet another compiler compiler," Comp.Sci.Tech,Rep No 32, Bell laboratories (July 1975)

[KER.83] A. Kerstenne,"Contribution a l'evaluation des performances d'algorithmes d'access"
Master degree thesis, FUNDP Namur.

[KERN.75] B.W. Kernighan "Ratfor -A preprocessor for a RATionnal FORtran," Software - Practice and experience, 5(1975),pp395-406

[KERN.72] B.W. Kernighan, D.M. Ritchie, and K. Thompson, "QED text editor," Comp.Sci.Tech,Rep No 39, Bell laboratories (May 1972)

[KNUTH.81] D.E. Knuth,"The art of computer programming," reading, MASS. : ADDISON-WESLEY 1981

[LESK.75] M.E. Lesk,"LEX -Alexical analyser generator," Comp.Sci.Tech,Rep No 39, Bell laboratories (October 1975)

[SEV.81] K.C. Sevcik,"Data base performance prediction using an analytical model"
Proc. 7th VLDB Conf. Cannes, France,
IEEE Computer Society

[STA.83] W. Staniszkis,"Probabilistic approach to evaluation of data manipulation algorithms in a Codasyl environment".
CRAI, Italy.

[TEO.83] T. Teorey,"Design of data base structures"
Wiley, New York, 1983.

[THOMPS.75] K. Thompson an D.M Ritchie, Unix programmer's manual, Bell laboratories, May 1975, section ed(1).

Facultes Universitaire Notre-Dame de la Paix (Namur)

Institut d'Informatique

CONTRIBUTION TO THE DESIGN
OF A DATABASE DESIGN
WORKBENCH

LOGICAL RECORD ACCESS EVALUATION

*** ANNEXES ***

Memoire presente par

Bruno DELCOURT,

Luc MOENTACK,

en vue de l'obtention
du titre de
licencie et maitre en informatique

Annee academique 1983 - 1984

Annexe 1 : Meta definition of SDLA (using ISLDM)

```
£ BRUNO DELCOURT                                    OCT 24 1983
£ LUC MOENTACK

£ SDLA.MTA
£
£ META DEFINITION OF Schema Description
£--------------------------------------------------------------
£
£
£               Schema  Description LAnguage (SDLA).
£               ===================================
£
£
£               At the logical layer, the designer works on an
£       abstract machine composed of a GAM DBMS and an ADL
£       processor. Moreover, the  LRA  evaluator  needs  a
£       statistic  description  of  the  data  structure.
£
£               The schema description language allows one to
£       create and maintain a data base which will contain all
£       the informations used at the logical layer.
£
£       With SDLA, the designer will be able to describe :
£
£       - The data structures manipulated at the logical layer
£       (namely, the PAS and the different NAS selected).
£
£       - The different predicative algorithms, and the
£       effective algorithms which will be derived.
£
£       - The statistic description of the data structure.
£
£--------------------------------------------------------------
£

KEYWORD DEFINE;
   SYNONYMS DEF;

KEYWORD SYNONYM;
   SYNONYMS SYN,SYNONYMS;

PREDEFINED-STATEMENT PREDEFINED-SECTION-STATEMENT;
   FORM DEFINE DEFINED-TYPE ( DEFINED-OBJECT : ,);

PREDEFINED-STATEMENT PREDEFINED-SYNONYM-STATEMENT;
   FORM SYNONYM ( SYNONYM-TYPE : ,) ;

PREDEFINED-STATEMENT PREDEFINED-TEXT-STATEMENT;
   FORM TEXT-TYPE ;


PREDEFINED-STATEMENT PREDEFINED-PROPERTY-STATEMENT;
   FORM PROPERTY-TYPE PROPERTY-VALUE-TYPE;
£
£ End of predefined part
£
£
£££££££££££££££££££££££££££££££££££££££££££££££££££££££££££


OPTIONAL-WORD IS;
OPTIONAL-WORD ARE;
```

££££££££££££££££££££££££££££££££££

```
NAME-CONSTANT REAL-FUNCTION;
DOCUMENTATION;
        Value of FUNCTION-TYPE property (see FUNC-DES).
        Means that the density function of the relation
        is completely known and that this function is given
        by an histogram;

NAME-CONSTANT UNIFORM-1;
DOCUMENTATION;
        Value of FUNCTION-TYPE property (see FUNC-DES).
        Means that the density function of the relation is
        an UNIFORM function;

NAME-CONSTANT UNIFORM-2;
DOCUMENTATION;
        Value of FUNCTION-TYPE property (see FUNC-DES).
        Means that the density function of the relation is
        an UNIFORM function;

NAME-CONSTANT UNIFORM-3;
DOCUMENTATION;
        Value of FUNCTION-TYPE property (see FUNC-DES).
        Means that the density function of the relation is
        an UNIFORM function;

NAME-CONSTANT UNIFORM-4;
DOCUMENTATION;
        Value of FUNCTION-TYPE property (see FUNC-DES).
        Means that the density function of the relation is
        an UNIFORM function;

NAME-CONSTANT POISSON;
DOCUMENTATION;
        Value of FUNCTION-TYPE property (see FUNC-DES).
        Means that the density function of the relation is
        a POISSON function;

NAME-CONSTANT BINOMIAL;
DOCUMENTATION;
        Value of FUNCTION-TYPE property (see FUNC-DES).
        Means that the density function of the relation is
        a BINOMIAL function;

NAME-CONSTANT GEOMETRIC-0;
DOCUMENTATION;
        Value of FUNCTION-TYPE property (see FUNC-DES).
        Means that the density function of the relation is
        a GEOMETRIC function;

NAME-CONSTANT GEOMETRIC-1;
DOCUMENTATION;
        Value of FUNCTION-TYPE property (see FUNC-DES).
        Means that the density function of the relation is
        a GEOMETRIC function;

NAME-CONSTANT DECREASING;
DOCUMENTATION;
        Value of FUNCTION-TYPE property (see FUNC-DES).
        Means that the density function of the relation is
        a DECREASING function;

NAME-CONSTANT MAX1-FUNCTION;
```

```
DOCUMENTATION;
        Value of FUNCTION-TYPE property (see FUNC-DES).
        Means that the density function of the relation is
        a MAXIMUM 1 function;

NAME-CONSTANT MAX2-FUNCTION;
DOCUMENTATION;
        Value of FUNCTION-TYPE property (see FUNC-DES).
        Means that the density function of the relation is
        a MAXIMUM 2 function;

NAME-CONSTANT J1-FUNCTION;
DOCUMENTATION;
        Value of FUNCTION-TYPE property (see FUNC-DES).
        Means that the density function of the relation is
        a J=1 function;

NAME-CONSTANT J2-FUNCTION;
DOCUMENTATION;
        Value of FUNCTION-TYPE property (see FUNC-DES).
        Means that the density function of the relation is
        a J=2 function;

NAME-CONSTANT REAL;
DOCUMENTATION;
        Value of TYPE-P (see STAT-DES).
        Means that the described ITEM is real;

NAME-CONSTANT INT;
DOCUMENTATION;
        Value of TYPE-P (see STAT-DES).
        Means that the described ITEM is integer;


NAME-CONSTANT ALPHA-NUM;
DOCUMENTATION;
        Value of TYPE-P (see STAT-DES).
        Means that the described ITEM is alphanumeric;

NAME-CONSTANT LIMITED-ALPHA-NUM;
SYNONYMS L-A-NUM;
DOCUMENTATION;
        Value of TYPE-P (see STAT-DES).
        Means that the described ITEM takes a value in a
        limited set of alphanumeric values;


NAME-CONSTANT YES;
SYNONYMS Y;

NAME-CONSTANT NO;
SYNONYMS N;

NAME-CONSTANT I-M;
DOCUMENTATION;
        Value of the FUNCTIONAL-CLASS property.
        ONE TO MANY;

NAME-CONSTANT I-I;
DOCUMENTATION;
        Value of the FUNCTIONAL-CLASS property.
        ONE TO ONE;

NAME-CONSTANT M-I;
```

```
DOCUMENTATION;
        Value of the FUNCTIONAL-CLASS property.
        MANY TO ONE;

NAME-CONSTANT M-N;
DOCUMENTATION;
        Value of the FUNCTIONAL-CLASS property.
        MANY TO MANY

NAME-CONSTANT IKO;
DOCUMENTATION;
        It indicates that the IKO represents :
                an identifier and
                a  key        and
                an order;

NAME-CONSTANT IK;
DOCUMENTATION;
        It indicates that the IKO represents :
                an identifier and
                a  key;

NAME-CONSTANT IO;
DOCUMENTATION;
        It indicates that the IKO represents :
                an identifier and
                an order;

NAME-CONSTANT I;
DOCUMENTATION;
        It indicates that the IKO represents :
                an identifier;

NAME-CONSTANT KO;
DOCUMENTATION;
        It indicates that the IKO represents :
                a  key        and
                an order;

NAME-CONSTANT K;
DOCUMENTATION;
        It indicates that the IKO represents :
                a  key;

NAME-CONSTANT O;
DOCUMENTATION;
        It indicates that the IKO represents :
                an order;

NAME-CONSTANT DB;
DOCUMENTATION;
        Value of REFERENCE-SET property ( see IKO ).
        DB = Database;

NAME-CONSTANT AP;
DOCUMENTATION;
        Value of REFERENCE-SET property ( see IKO ).
        AP = Access path type;

NAME-CONSTANT EF;
DOCUMENTATION;
        Value of REFERENCE-SET property ( see IKO ).
        EF = Each file;
```

```
NAME-CONSTANT LIFO;
SYNONYMS L;
DOCUMENTATION;
        Value of DUPLICATES property ( see IKO ).;


NAME-CONSTANT FIFO;
SYNONYMS F;
DOCUMENTATION;
        Value of DUPLICATES property ( see IKO ).;


NAME-CONSTANT RANDOM;
SYNONYMS R;
DOCUMENTATION;
        Value of DUPLICATES property ( see IKO ).;

NAME-CONSTANT PAS-NAS;
DOCUMENTATION;
        Value of TYPE-P property of the TRANSFORMATION-OF
        relationship;

NAME-CONSTANT PRED-EFF;
DOCUMENTATION;
        Value of TYPE-D property of the DERIVED-FROM
        relationship;


££££££££££££££££££££££££££££££££££

VALUE-RANGE SEQ-RANGE;
   VALUES INTEGER;

VALUE-RANGE ORD-RANGE;
   VALUES STRING;

VALUE-RANGE TYPE-T-RANGE;
        VALUES PAS-NAS;

VALUE-RANGE TYPE-D-RANGE;
        VALUES PRED-EFF;

££££££££££££££££££££££££££££££££££

£-------------------------------------------------------------

OBJECT ALGORITHM;
SYNONYM ALG;
DOCUMENTATION;
        1)It represents an effective algorithm. The algorithm is
        written in ADL and use the objects that "its" NAS allows
        it to use.
        This algorithm is derived from a predicative algorithm.
        Many effective algorithms can be derived from one
        predicative algorithm.

        2)It represent a predicative algorithm. The algorithm is
        written in ADL an use the object that "its" PAS allows it
        to use.
        From one predicative algorithm, one can derive many
        effective algorithms.;
```

```
CODE OBALG 11;

PROPERTY ALGORITHM-TYPE;
SYNONYM ALG-TYPE;
APPLIES ALG;
DOCUMENTATION;
        Indicates the type of the algorithm : - EFFECTIVE
                                              - PREDICATIVE;
VALUES EFFECTIVE, PREDICATIVE;
CODE PYALGT 108;

TEXT ALGORITHM-TEXT;
SYNONYM ALG-TEXT;
APPLIES ALG;
DOCUMENTATION;
        Text of the algorithm, in ADL;
CODE TXALGE 203;

£----------------------------------------------------------

OBJECT SCHEMA;
DOCUMENTATION;
        It represents a proposal for a DB schema.

        1) It represents the possible accesses schema (NAS).
        The PAS is the result of the systematic translation of
        the E-R schema defined at the conceptual level.
        Basically, it allows one to perform all accesses to
        the data (each item may be used as an access , each
        path has an inverse).
        We use the GAM (Generalized Access Model) to define the
        PAS. The GAM is a convenient way to express the data
        structure in term of ITEM, RECORD-TYPE,and ACCESS PATH
        TYPE.

        2) It represents the necessary accesses schema (NAS).
        The NAS represents a subset of the PAS. Usually, each
        algorithm does not needs all the data structures
        specified in the PAS. The NAS expresses the data
        structures that the associated algorithms need to work.
        The integration of all the NAS for one application
        represent the data structure that the application need
        to work. The integrated NAS is an output of the logical
        level. So one NAS contains all the records type of the
        associated PAS, but it selects the access path type and
        the IKO it needs to work, and only the ones it needs.;

CODE OBSCH 1;

PROPERTY SCHEMA-TYPE;
DOCUMENTATION;
        Indicates the type of the schema : - NAS
                                           - PAS;
APPLIES SCHEMA;
VALUES NAS,PAS;
CODE PYSCHT 112;

£----------------------------------------------------------

OBJECT FILE;
DOCUMENTATION;
        It represents a file associated to the SCHEMA. A file is
        a set of records.  A file may collects more than one
        record type. But a file belongs to one and only one
```

```
        SCHEMA.;

CODE OBFIL 2;

£-------------------------------------------------------------

OBJECT RECORD-TYPE;
SYNONYMS RECORD;
DOCUMENTATION;
        It represents a record type. A record type represents
        the existence of one kind of records. The object RECORD
        has no property. The caracteristics of the record type
        are expressed via the property of the other objects
        which are related to it.

                these objects may be :

                - The file(s) wich contain(s) the record type.
                - The item(s) contained in the record type.
                - The path(s) wich use(s) the record type as a
                target or as an origin.
                - The IKO that refer(s) to the record type.;

CODE OBREC 3;

£-------------------------------------------------------------

OBJECT ITEM;
DOCUMENTATION;
        It represents a data item contained in a record type
        or decomposable data item.

        e.g: 1) an item  NAME  of a record  EMPLOYEE.

             2) An item  CITY  of the decomposable item
                ADDRESS  of a record  EMPLOYEE.;

CODE OBIT 4;

PROPERTY OPTIONAL;
APPLIES ITEM;
DOCUMENTATION;
        It indicates if a data item is optional or not;
VALUES YES,NO;
CODE PYOPT 102;

PROPERTY REPETITIVITY;
SYNONYMS REPET;
APPLIES ITEM;
DOCUMENTATION;
        It indicates if a data item is repetitive or not;
VALUES YES,NO;
CODE PYREP 103;

TEXT FORMAT;
APPLIES ITEM;
DOCUMENTATION;
        Description of the format of a data item;
CODE TXFT 201;

£-------------------------------------------------------------

OBJECT ACCESS-PATH-TYPE;
SYNONYMS PATH;
```

```
DOCUMENTATION;
        It represents an acces path type. The access path
        is a way to access from a record occurrence (which
        is called 'ORIGIN') to one or many record
        occurrences (which is/are called 'TARGET(S)'). The
        access path type represents one type of access path;

CODE OBAPT 5;

PROPERTY FUNCTIONAL-CLASS;
APPLIES PATH;
DOCUMENTATION;
        It indicates the functional-class of a PATH
                I-I             ONE  ORIGIN     ONE  TARGET
                I-M             ONE  ORIGIN     MANY TARGETS
                M-I             MANY ORIGINS    ONE  TARGET
                M-N             MANY ORIGINS    MANY TARGETS;

VALUES I-M,I-I,M-I,M-N;
CODE PYCON 104;

£---------------------------------------------------------------

OBJECT IKO;
DOCUMENTATION;
        a) The IKO is the generalization of three concepts :

                1) The concept of identifier.
                =============================

                Thru concept of identifier, we want to
        express the fact that a group of item is identifier
        for a record type. In other words, there is a
        functional dependency between the identifier and the
        item(s) of the record involved. In this case, an
        occurrence of an IKO expresses the fact that a group
        of item is identifier for a record type.

        The domain of validity of the identifier may be :

        - All the data base.
        - Each file which contains the record, but each file
        taken separatly.
        - Within one or many particular access path type.


                2) The concept of key.
                ======================

                Thru the concept of key, we want to express
        the fact that a group of items is used as an access
        key. In this case, one occurrence of an IKO expresses
        the fact that a group of item is used as an access
        key.


                3) The concept of order.
                ========================

                Thru the concept of order, we want to
        express the fact that a set of record is sorted. The
        "sort key" may be a group of item. In this case, an
        occurrence of an IKO designates the group of item
        wich is the sort key for one type of records.
```

The domain of validity of the order may be :

- All the data base.
- Each file which contains the record, but each file taken separatly.
- Within one or many particular access path type.

b) An occurrence of an IKO may express one, two or three concepts at the same time.

c) The IKO may belong to a GLOBAL. Thru the concept of global, we want to extend the domain of validity of an identifier, a key, a order to many record types. There are items in different record types, which contain the same kind of information (for example worker_number and employee_number are both number of persons). If there is no intersection between the sets of possible values (worker_number takes values from 1 to 1000, and employee_number takes values from 1001 to 2000), we can say that both numbers are identifiers for both record types.

To express this fact, we will imagine a global (for example, GLOBAL 1) and we will say that both IKO belong to the global. (for both IKO we will have the property GLOBAL with value 1).;

CODE OBIKO 6;

PROPERTY GLOBAL;
APPLIES IKO;
DOCUMENTATION;
If 0 $=^3$ not global
If n $=^3$ number of the global to which the IKO belongs;
VALUES INTEGER 0 THRU 1000;
CODE PYGLO 105;

PROPERTY REFERENCE-SET;
SYNONYMS REF-SET;
APPLIES IKO;
DOCUMENTATION;
Indicates the reference set type.
DB = All the DB schema.
EF = Each file taken separatly.
AP = The path types related to the IKO;
VALUES DB,EF,AP;
CODE PYREF 106;

PROPERTY IDENT-KEY-ORDER;
SYNONYMS ID-K-O;
APPLIES IKO;
DOCUMENTATION;
Indicates if an IKO represents an identifier, a key, or an order
E.G - IK = identifier AND key AND NOT order
- KO = NOT identifier AND key AND order;
VALUES IKO,IK,IO,I,K,KO,O;
CODE PYIDE 107;

PROPERTY DUPLICATES;
SYNONYMS DUP;
APPLIES IKO;
DOCUMENTATION;

```
        Indicates if an access key allows duplicates.
        And gives the order of storage of the duplicates
        if any.
                NO : no duplicates
                LIFO
                FIFO
                RANDOM;
VALUES NO,L,F,R;
CODE PYDUP 110;

TEXT ORDER-TYPE;
APPLIES  IKO;
DOCUMENTATION;
        Description of the order of the records.;
CODE TXORD 202;

£-------------------------------------------------------------

OBJECT STATISTIC-DESCRIPTION;
SYNONYMS STAT-DES;
DOCUMENTATION;
        It represents the statistic description of one and
        only one of the following object : record, item or
        access path.;

CODE OBSTAT 7;

PROPERTY TYPE-P;
APPLIES STAT-DES;
DOCUMENTATION;
        Gives the type of proportion if any;
VALUES REAL,INT,ALPHA-NUM,LIMITED-ALPHA-NUM;
CODE PYTYP 129;

PROPERTY NUM-ELEMENTS;
APPLIES STAT-DES;
DOCUMENTATION;
        If STAT-DES describes a record type, it gives the
        number of occurrences of the record type in the
        database.;
VALUES INTEGER;
CODE PYNEL 160;

£-------------------------------------------------------------

OBJECT SEGMENT;
DOCUMENTATION;
        It represents the description of one class of the
        F(k) distribution  function.
        The F(k) describes the relation between two objects
        r(A,B). The objects may be record types or items.;


CODE OBSEG 8;

PROPERTY MAX-CLASS-S;
SYNONYMS M-CL-S;
APPLIES SEGMENT;
DOCUMENTATION;
        Gives the upper limit of the class represented by
        the segment;
VALUES INTEGER;
CODE PYMAXCS 130;
```

```
PROPERTY VALUE-S;
APPLIES SEGMENT;
DOCUMENTATION;
        Gives the proportion of occurrences of objects which
        belong to the class described by the segment;
VALUES NUMBER 0.0 THRU 1.0;
CODE PYVALS 131;

£------------------------------------------------------------

OBJECT PROPORTION;
DOCUMENTATION;
        It represents the description of one class of the
        proportion function used for description the
        distribution of the values of one non decomposable
        data item;

CODE OBPRO 9;

PROPERTY MAX-CLASS-P;
SYNONYMS M-CL-P;
APPLIES PROPORTION;
DOCUMENTATION;
        Gives the upper limit of the class represented by
 value of the described ITEM
        in that class;
VALUES NUMBER 0.0 THRU 1.0;
CODE PYVALP 133;

£------------------------------------------------------------

OBJECT FUNCTION-DESCRIPTION;
SYNONYMS FUNC-DES;
DOCUMENTATION;
        It represents the statistic description of the direct
        or inverse function;
CODE OBFUNC 10;

PROPERTY FUNCTION-TYPE;
APPLIES FUNC-DES;
DOCUMENTATION;
        Indicates the type of density function used to
        describe the relation;
VALUES REAL-FUNCTION,UNIFORM-1,UNIFORM-2,UNIFORM-3,UNIFORM-4,
       POISSON,BINOMIAL,GEOMETRIC-0,GEOMETRIC-1,DECREASING,
       MAX1-FUNCTION,MAX2-FUNCTION,J1-FUNCTION,J2-FUNCTION;
CODE PYTYP 111;

PROPERTY MEAN;
APPLIES FUNC-DES;
DOCUMENTATION;
        Gives the average of the function;
VALUES NUMBER;
CODE PYMEAN 112;

PROPERTY F-ZERO;
APPLIES FUNC-DES;
DOCUMENTATION;
        Gives the value in zero of the function;
VALUES NUMBER 0.0 THRU 1.0;
CODE PYFZ 113;

PROPERTY KMAX;
APPLIES FUNC-DES;
```

```
DOCUMENTATION;
        Gives the value of k where the function is maximum;
VALUES NUMBER;
CODE PYKMAX 114;

PROPERTY PAR-ONE;
APPLIES FUNC-DES;
DOCUMENTATION;
        Parameter one of the function;
VALUES NUMBER;
CODE PYPO 115;

PROPERTY PAR-TWO;
APPLIES FUNC-DES;
DOCUMENTATION;
        Parameter two of the function;
VALUES NUMBER;
CODE PYPT 116;

PROPERTY MIN-RELATIONS;
SYNONYM MIN-R;
APPLIES FUNC-DES;
DOCUMENTATION;
        Minimum number of relation in which the object is
        involved;
VALUES INTEGER;
CODE PYI 117;

PROPERTY MAX-RELATIONS;
SYNONYM MAX-R;
APPLIES FUNC-DES;
DOCUMENTATION;
        Maximum number of relation in which the object is
        involved;
VALUES INTEGER;
CODE PYJ 118;

PROPERTY NUMBER-TARGET;
SYNONYM NT;
APPLIES FUNC-DES;
DOCUMENTATION;
        The function describes a relation r(A,B) between two
        sets of objects.  NT is the number of B objects;
VALUES INTEGER;
CODE PYNT 119;

£------------------------------------------------------------

££££££££££££££££££££££££££££££££££

£------------------------------------------------------------
RELATIONSHIP FILE-OF-RSH;
DOCUMENTATION ;
        It indicates that a file belongs to a SCHEMA;
PARTS FS-FILE,FS-SCH;
COMBINATION FS-FILE FILE
WITH FS-SCH SCHEMA;
CONNECTIVITY ONE FS-SCH MANY FS-FILE;
CODE RTFS 301;
CONNECTION-TYPE S2;
STORED FS-SCH 1,FS-FILE 2;

STATEMENT FILE-SCH-ST;
DOCUMENTATION;
```

```
        Used to specify the SCHEMA to which the file belongs;
USED FS-FILE FILE-OF-RSH;
FORM FILE-OF FS-SCH;

£-------------------------------------------------------------

RELATIONSHIP COLLECTS-RSH;
DOCUMENTATION ;
        It indicates that one or many file contains one or
        many record types;
PARTS FR-FILE,FR-RECORD;
COMBINATION FR-FILE FILE
WITH FR-RECORD RECORD;
CONNECTIVITY MANY FR-FILE MANY FR-RECORD;
CODE RTFR 303;
CONNECTION-TYPE S1;
STORED FR-FILE 1,FR-RECORD 2;

STATEMENT COLLECTS-ST;
DOCUMENTATION;
        Used to specify the record types collected in the
        file;
USED FR-FILE COLLECTS-RSH;
FORM COLLECTS (FR-RECORD:,);

STATEMENT COLLECTED-IN-ST;
DOCUMENTATION;
        Used to specify the files which contain the record
        type;
USED FR-RECORD COLLECTS-RSH;
FORM COLLECTED IN (FR-FILE:,);

£-------------------------------------------------------------

RELATIONSHIP INVERSE-OF-RSH;
DOCUMENTATION ;
        It indicates that a path type is inverse of another
        one;
PARTS INV-O,INV-T;
COMBINATION INV-O PATH
WITH INV-T PATH;
CONNECTIVITY ONE INV-O ONE INV-T;
CODE RTINV 304;
CONNECTION-TYPE S4;
STORED INV-O 1,INV-T 2;

STATEMENT INVERSE-ST;
DOCUMENTATION;
        Used to specify the inverse path type of a path type;
USED INV-O INVERSE-OF-RSH;
FORM INVERSE IS INV-T;

£-------------------------------------------------------------

RELATIONSHIP CONTAINED-RSH;
DOCUMENTATION ;
        It indicates that an item is contained in either a
        record type, or another item;
PARTS CONT-REC-ITEM , CONT-ITEM;
COMBINATION CONT-REC-ITEM RECORD,ITEM
WITH CONT-ITEM ITEM;
CONNECTIVITY MANY CONT-ITEM ONE CONT-REC-ITEM;
CODE RTCONT 307;
CONNECTION-TYPE S2;
```

```
STORED CONT-REC-ITEM 1,CONT-ITEM 2;

STATEMENT CONTAINED-IN-ST;
DOCUMENTATION;
        Used to specify the record type or the item to which
        an item belongs;
USED CONT-ITEM CONTAINED-RSH;
FORM CONTAINED IN CONT-REC-ITEM;

£-------------------------------------------------------------

RELATIONSHIP CONCERNS-RSH;
DOCUMENTATION ;
        It indicates that an IKO refers to a record type;
PARTS RI-RECORD,RI-IKO;
COMBINATION RI-RECORD RECORD
WITH RI-IKO IKO;
CONNECTIVITY ONE RI-RECORD MANY  RI-IKO;
CODE RTRI 308;
CONNECTION-TYPE S2;
STORED RI-RECORD 1,RI-IKO 2;

STATEMENT CONCERNS-ST;
DOCUMENTATION;
        Used to specify the record type which is the
        reference of the IKO;
USED RI-IKO CONCERNS-RSH;
FORM CONCERNS RI-RECORD;

£-------------------------------------------------------------

RELATIONSHIP WITHIN-RSH;
DOCUMENTATION ;
        It indicates that the acces path type belongs to the
        reference set of an IKO;
PARTS IP-IKO,IP-PATH;
COMBINATION IP-IKO IKO
WITH IP-PATH PATH;
CONNECTIVITY MANY IP-PATH MANY IP-IKO;
CODE RTIP 309;
CONNECTION-TYPE S1;
STORED IP-IKO 1,IP-PATH 2;

STATEMENT WITHIN-ST;
DOCUMENTATION;
        Used to specify the access path types which constitute
        the reference set of the IKO;
USED IP-IKO WITHIN-RSH;
FORM WITHIN (IP-PATH:,);

£-------------------------------------------------------------

RELATIONSHIP CONSISTS-OF-RSH;
DOCUMENTATION ;
        It indicates that one item is one of the constituents
        of one IKO.
        ORD and SEQ are properties of the relationship.
        ORD = A if the order is ascending on the item.
            = D if the order is descending on the item.
            = N if the IKO is not an order.
        SEQ is a sequence number. If there is more than one
        item, the sort key is sorted on the sequence number
        (ascending).
```

```
        e.g: last_name with seq = 1 and first_name with
        seq = 2, means that the record type PEOPLE is sorted
        first on the last_name and then on the first_name.;
PARTS CON-IKO,CON-ITEM,CON-SEQ,CON-ORD;
COMBINATION CON-IKO IKO
WITH CON-ITEM ITEM
WITH CON-SEQ VALUE-FOR SEQ-RANGE
WITH CON-ORD VALUE-FOR ORD-RANGE;
CONNECTIVITY MANY CON-IKO,CON-ITEM,CON-SEQ,CON-ORD;
CODE RTCONS 310;
CONNECTION-TYPE F1;
STORED CON-IKO 1,CON-ITEM 2,CON-SEQ 3,CON-ORD 4;

STATEMENT CONSISTS-OF-ST;
DOCUMENTATION;
        Used to specify the items which constitute the IKO;
USED CON-IKO CONSISTS-OF-RSH;
FORM CONSISTS OF (CON-ITEM [WITH SEQ CON-SEQ ORD CON-ORD] :,);

£-------------------------------------------------------------

RELATIONSHIP COUNTED-RSH;
DOCUMENTATION ;
        It indicates that an item is the repetitivity counter
        of a repetitive item;
PARTS COUNT-ITEM,COUNTED-ITEM;
COMBINATION COUNT-ITEM ITEM
WITH COUNTED-ITEM ITEM;
CONNECTIVITY ONE COUNT-ITEM MANY COUNTED-ITEM;
CODE RTCT 311;
CONNECTION-TYPE S2;
STORED COUNT-ITEM 1,COUNTED-ITEM 2;

STATEMENT COUNTED-ST;
DOCUMENTATION;
        Used to specify the counter of the repetitive item;
USED COUNTED-ITEM COUNTED-RSH;
FORM COUNTED BY COUNT-ITEM;

£-------------------------------------------------------------

RELATIONSHIP TARGET-RSH;
DOCUMENTATION ;
        It indicates  what is/are the type(s) of
        occurrences of records  that may be target of one
        access path type.;
PARTS T-PATH,T-TARGET;
COMBINATION T-PATH PATH
WITH T-TARGET RECORD;
CONNECTIVITY MANY T-PATH,T-TARGET;
CODE RTTAR 305;
CONNECTION-TYPE S1;
STORED T-PATH 1,T-TARGET 2;

STATEMENT TARG-ST;
DOCUMENTATION;
        Used to specify the targets of an access path
        type;
USED T-PATH TARGET-RSH;
FORM TARGET (T-TARGET:,);

£-------------------------------------------------------------

RELATIONSHIP ORIGIN-RSH;
```

```
DOCUMENTATION ;
        It indicates  what is/are the type(s) of occurrences
        of records that may be ORIGIN of one access path type.;
PARTS T-PATH,T-ORIGIN;
COMBINATION T-PATH PATH
WITH T-ORIGIN RECORD;
CONNECTIVITY MANY T-PATH,T-ORIGIN;
CODE RTOR 312;
CONNECTION-TYPE S1;
STORED T-PATH 1,T-ORIGIN 2;

STATEMENT ORIG-ST;
DOCUMENTATION;
        Used to specify the origins of an access path
        type;
USED T-PATH ORIGIN-RSH;
FORM ORIGIN (T-ORIGIN:,);

£-------------------------------------------------------------

RELATIONSHIP DERIVED-FROM-RSH;
DOCUMENTATION;
        It indicates that an effective algorithm is derived
        from a predicative algorithm.
        TYPE-D gives the type of derivation;
PARTS ALG-PRED,ALG-EFF,TY-D
COMBINATION ALG-PRED ALG
WITH ALG-EFF ALG
WITH TY-D VALUE-FOR TYPE-D-RANGE
CONNECTIVITY ONE ALG-PRED MANY ALG-EFF,TY-D
CODE RTDF 313;
CONNECTION-TYPE G3;
STORED ALG-PRED 1,ALG-EFF 2,TY-D 3;

STATEMENT DERIVED-FROM-ST;
DOCUMENTATION;
        Used to specify the predicative algorithm from which
        the effective algorithm is derived;
USED ALG-EFF DERIVED-FROM-RSH;
FORM DERIVED FROM ALG-PRED WITH TYPE-D TY-D;

£-------------------------------------------------------------

RELATIONSHIP TRANSFORMATION-RSH;
DOCUMENTATION;
        It indicates that a SCHEMA is the transformation
        of another SCHEMA.
        eg. a NAS is the transformation of a PAS.
        TYPE-T gives the type of transformation;

PARTS PAS,NAS,TY-T;
COMBINATION PAS SCHEMA
WITH NAS SCHEMA
WITH TY-T VALUE-FOR TYPE-T-RANGE;
CONNECTIVITY ONE PAS MANY NAS,TY-T
CODE RTSUBS 314;
CONNECTION-TYPE G3;
STORED PAS 1,NAS 2,TY-T 3;

STATEMENT TRANSFORMATION-OF-ST;
DOCUMENTATION;
        Used to specify the schema of which another schema
        is the transformation.;
USED NAS TRANSFORMATION-RSH;
```

```
FORM TRANSFORMATION OF PAS;

STATEMENT SET-OF-ST;
DOCUMENTATION;
        Used to specify the NAS which are subset of the PAS;
USED PAS SUBSET-RSH;
FORM SET OF (NAS:,);

£-------------------------------------------------------------

RELATIONSHIP WORKS-ON-RSH;
DOCUMENTATION;
        It indicates that an algorithm works on a schema;
PARTS WO-SCHEMA,WO-ALG;
COMBINATION WO-SCHEMA SCHEMA
WITH WO-ALG ALG;
CONNECTIVITY ONE WO-SCHEMA MANY WO-ALG;
CODE RTWO 321;
CONNECTION-TYPE S2;
STORED WO-SCHEMA 1,WO-ALG 2;

STATEMENT SCHEMA-ALG-ST;
DOCUMENTATION;
        Used to specify the SCHEMA on which the algorithm
        works;
USED WO-ALG WORKS-ON-RSH;
FORM WORKS ON WO-SCHEMA;

£-------------------------------------------------------------

RELATIONSHIP RECORD-OF-RSH;
DOCUMENTATION ;
        It indicates that a RECORD belongs to a SCHEMA;
PARTS RS-RECORD,RS-SCH;
COMBINATION RS-RECORD RECORD
WITH RS-SCH SCHEMA;
CONNECTIVITY ONE RS-SCH MANY RS-RECORD;
CODE RTRS 316;
CONNECTION-TYPE S2;
STORED RS-SCH 1,RS-RECORD 2;

STATEMENT RECORD-SCH-ST;
DOCUMENTATION;
        Used to specify the SCHEMA to which the RECORD belongs;
USED RS-RECORD RECORD-OF-RSH;
FORM RECORD-OF RS-SCH;

£-------------------------------------------------------------

RELATIONSHIP DIRECT-FUNCTION-OF-RSH;
DOCUMENTATION ;
        It indicates that the function description belongs to
        the statistic description and that it describes one
        class of the direct function;

PARTS DIR-STAT-DES,DIR-FD;
COMBINATION DIR-STAT-DES STAT-DES
WITH DIR-FD FUNC-DES;
CONNECTIVITY ONE DIR-STAT-DES,DIR-FD;
CODE RTDIR 312;
CONNECTION-TYPE S4;
STORED DIR-STAT-DES 1,DIR-FD 2;

STATEMENT DIRECT-ST;
```

```
DOCUMENTATION;
        Used to specify that the function description
        describes the  "direct"  relation;
USED DIR-STAT-DES DIRECT-FUNCTION-OF-RSH;
FORM DIRECT FUNCTION IS DIR-FD;

£-------------------------------------------------------------

RELATIONSHIP INVERSE-FUNCTION-OF-RSH;
DOCUMENTATION ;
        It indicates that the function description belongs to
        the statistic description and that it describes one
        class of the inverse function;

PARTS INV-DES-SD,INV-DES-FD;
COMBINATION INV-DES-SD STAT-DES
WITH  INV-DES-FD FUNC-DES;
CONNECTIVITY ONE INV-DES-SD,INV-DES-FD;
CODE RTINDES 313;
CONNECTION-TYPE S4;
STORED INV-DES-SD 1,INV-DES-FD 2;

STATEMENT INV-DES-ST;
DOCUMENTATION;
        Used to specify that the function description
        describes the "inverse" relation;
USED INV-DES-SD INVERSE-FUNCTION-OF-RSH;
FORM INVERSE FUNCTION IS INV-DES-FD;

£-------------------------------------------------------------

RELATIONSHIP DESCRIBE-PATH-RSH;
DOCUMENTATION ;
        It indicates that the statistic description describes
        an access path type;
PARTS DESCR-PATH, DESCR-SD;
COMBINATION DESCR-PATH PATH
WITH DESCR-SD STAT-DES;
CONNECTIVITY ONE DESCR-PATH,DESCR-SD;
CODE RTDESCR 314;
CONNECTION-TYPE S4;
STORED DESCR-PATH 1,DESCR-SD 2;

STATEMENT DESCRIBES-PATH-ST;
DOCUMENTATION;
        Used to specify the access path type described by the
        statistic description;
USED DESCR-SD DESCRIBE-PATH-RSH;
FORM DESCRIBES-PATH DESCR-PATH;

£-------------------------------------------------------------

RELATIONSHIP DESCRIBE-ITEM-RSH;
DOCUMENTATION ;
        It indicates that the statistic description describes
        an item;
PARTS DEF-SD,DEF-ITEM;
COMBINATION DEF-SD STAT-DES
WITH DEF-ITEM ITEM;
CONNECTIVITY ONE DEF-SD,DEF-ITEM;
CODE RTDEF 315;
CONNECTION-TYPE S4;
STORED DEF-ITEM 1,DEF-SD 2;
```

```
STATEMENT DESCRIBES-ITEM-ST;
DOCUMENTATION;
        Used to specify the item decribed by the statistic
        description;
USED DEF-SD DESCRIBE-ITEM-RSH;
FORM DESCRIBES-ITEM DEF-ITEM;

£-------------------------------------------------------------
RELATIONSHIP DESCRIBE-RECORD-RSH;
DOCUMENTATION ;
        It indicates that the statistic description describes
        a record type
PARTS DEF-SD,DEF-RECORD;
COMBINATION DEF-SD STAT-DES
WITH DEF-ITEM RECORD;
CONNECTIVITY ONE DEF-SD,DEF-RECORD;
CODE RTDEF 350;
CONNECTION-TYPE S4;
STORED DEF-RECORD 1,DEF-SD 2;

STATEMENT DESCRIBES-RECORD-ST;
DOCUMENTATION;
        Used to specify the record decribed by the statistic
        description;
USED DEF-SD DESCRIBE-RECORD-RSH;
FORM DESCRIBES-RECORD DEF-RECORD;

£-------------------------------------------------------------

RELATIONSHIP PROPORTION-OF-RSH;
DOCUMENTATION ;
        It indicates that the proportion belongs to the
        statistic description of an item;
PARTS PROP-SD,PROP-PROP;
COMBINATION PROP-SD STAT-DES
WITH PROP-PROP PROPORTION;
CONNECTIVITY ONE PROP-SD MANY PROP-PROP;
CODE RTPROP 316;
CONNECTION-TYPE S2;
STORED PROP-SD 1,PROP-PROP 2;

STATEMENT PROPORTION-ST;
DOCUMENTATION;
        Used to specify the statistic description a
        proportion belongs to;
USED PROP-PROP PROPORTION-OF-RSH;
FORM PROPORTION-OF PROP-SD;

£-------------------------------------------------------------

RELATIONSHIP SEGMENT-OF-RSH;
DOCUMENTATION;
        It indicates that a segment belongs to the description
        of the function;
PARTS SEG-SEG,SEG-FD;
COMBINATION SEG-SEG SEGMENT
WITH SEG-FD FUNC-DES;
CONNECTIVITY ONE SEG-FD MANY SEG-SEG;
CODE RTSEG 317;
CONNECTION-TYPE S2;
STORED SEG-FD 1,SEG-SEG 2;

STATEMENT SEGMENT-OF-ST;
DOCUMENTATION;
```

```
        Used to specify the function description to which the
        segment belongs;
USED SEG-SEG SEGMENT-OF-RSH;
FORM SEGMENT-OF SEG-FD;

STATEMENT SEGMENTS-ARE-ST;
DOCUMENTATION;
        Used to specify the segments describing a
        real function;
USED SEG-FD SEGMENT-OF-RSH;
FORM SEGMENTS-ARE SEG-SEG;

££££££££££££££££££££££££££££££££££££££££££
```

Annexe 2 : Lexical definition of ADL (using LEX)

```
/*      Lexical part of ADL.
        --------------------
*/

letter          [A-Za-z]

sign            [+-]

digit           [0-9]

unsinteger      {digit}+

decnumber       ({unsinteger}"."{unsinteger}

                |"."{unsinteger}|{unsinteger}".")

exppart         ("E"{unsinteger})

unsreal         ({decnumber}({exppart}?))

string          '[^\'\n]*'

boolean         TRUE|FALSE|true|false

name            {letter}({letter}|_|{digit})*

relop           ("<"|">"|"=<"|">="|"="|"<>"|"in"|"not_in")

plus            "+"

minus           "-"

mult            "*"

div             "/"

assign          ":="

exp             "\^"

excla           "!"

colon           ":"

bracketo        "["

bracketc        "]"

crossroad       "#"
```

```
comma           ","

period          "."

senicolon       ";"

parenthesiso    "("

parenthesisc    ")"

braceo          "{"

bracec          "}"

conments        "/*"[^("*/")]*"*/"

others          [^" "\n\t]

nul             [" "\n\t]+


%%
algorithm       return(T_ALG)    ;

algs            return(T_ALGS)   ;

var             return(T_VAR)    ;

const           return(T_CONST)  ;

boolean         return(T_BOOL)   ;

string          return(T_STR)    ;

real            return(T_REAL)   ;

integer         return(T_INT);

record          return(T_REC);

funcs           return(T_FUNCS);

structure       return(T_STRUCTURE);

end             return(T_END)    ;

endfor          return(T_ENDFOR)         ;

endwhile        return(T_ENDWHILE)       ;

endif           return(T_ENDIF) ;
```

```
array           return(T_ARRAY) ;
of              return(T_OF)    ;
begin           return(T_BEGIN) ;
not             return(T_NOT)   ;
for             return(T_FOR)   ;
do              return(T_DO)    ;
order           return(T_ORDER) ;
ascending       return(T_ASCENDING)     ;
descending      return(T_DESCENDING)    ;
modify          return(T_MODIFY)        ;
create          return(T_CREATE)        ;
delete          return(T_DELETE)        ;
attach          return(T_ATTACH)        ;
detach          return(T_DETACH)        ;
transfer        return(T_TRANSFER)      ;
to              return(T_TO)    ;
from            return(T_FROM)  ;
via             return(T_VIA)   ;
if              return(T_IF)    ;
then            return(T_THEN)  ;
else            return(T_ELSE)  ;
while           return(T_WHILE) ;
and             return(T_AND)   ;
or              return(T_OR)    ;
call            return(T_CALL)  ;
list            return(T_LIST)  ;
schema          return(T_SCHEMA)  ;
exit            return(T_EXIT)  ;
next            return(T_NEXT)  ;
```

```
{comments}          ;

{braceo}            return(T_BRACEO)   ;

{bracec}            return(T_BRACEC)   ;

{boolean}           return(T_BOOL_K)   ;

{relop}             return(T_RELOP) ;

{name}              return(T_IDENTIFIER)      ;

{unsreal}           return(T_REAL_K);

{unsinteger}        return(T_INT_K) ;

{string}            { int i;
        for (i=0; i<(yyleng-1);(*(yytext + i -1) = *(yytext + ++i)));
                    yyleng=yyleng-2;
                    return(T_STR_K) ; }

{nul}               ;

{comma}             return(T_COMMA) ;

{period}            return(T_PERIOD)          ;

{assign}            return(T_ASSIGN);

{semicolon}         return(T_SEMICOLON)       ;

{colon}             return(T_COLON);

{plus}              return(T_PLUS);

{minus}             return(T_MINUS);

{mult}              return(T_MULT);

{div}               return(T_DIV);

{exp}               return(T_EXP);

{excla}             return(T_EXCLA);

{bracketo}          return(T_BRAK_O);

{bracketc}          return(T_BRAK_C);

{crossroad}         return(T_CROSS);

{parenthesiso} return(T_PAR_O);

{parenthesisc} return(T_PAR_C);

{others}            return(T_OTHERS)          ;

%%
yywrap(){
        return(1);
        }
```

Annexe 3 : Syntactical definition of ADL (using YACC)

```
%{
# include   <stdio.h>
# include   <ctype.h>
# define    t_int_k        -1
# define    t_str_k        -2
# define    t_alg          -5
# define    t_alh          -6
# define    t_alb          -25
# define    t_exdp         -7
# define    t_algdec       -11
# define    t_funcdec      -12
# define    t_felt         -13
# define    t_vadp         -16
# define    t_vardec       -17
# define    t_ident        -18
# define    t_bool         -19
# define    t_str          -20
# define    t_real         -21
# define    t_int          -22
# define    t_rectyp       -23
# define    t_array        -24
# define    t_struc        -26
# define    t_names        -27
# define    t_indexs       -28
# define    t_genexp       -35
# define    t_orexp        -36
# define    t_andexp       -37
# define    t_not          -38
# define    t_tesexp       -39
# define    t_arexp        -40
# define    t_mulexp       -42
# define    t_exexp        -43
# define    t_plus         -44
# define    t_minus        -45
# define    t_mult         -46
# define    t_div          -47
# define    t_relop        -49
# define    t_belcon       -50
# define    t_var          -51
# define    t_arrelt       -52
# define    t_struct       -53
# define    t_unsnum       -54
# define    t_logval       -56
# define    t_co           -57
# define    t_tesop        -58
# define    t_ord          -59
# define    t_card         -60
# define    t_relcon       -63
# define    t_subls        -64
# define    t_stp          -65
# define    t_enumloop     -66
# define    t_forlis       -67
# define    t_order        -68
# define    t_ordkey       -69
# define    t_asc          -70
```

```
# define   t_des         -71
# define   t_nad         -72
# define   t_dbmod       -73
# define   t_modify      -74
# define   t_creat       -75
# define   t_del         -76
# define   t_att         -77
# define   t_det         -78
# define   t_transf      -79
# define   t_altern      -80
# define   t_whileloop   -81
# define   t_call        -82
# define   t_assst       -83
# define   t_func        -86
# define   t_parlis      -87
# define   t_schdec      -88
# define   t_listyp      -89
# define   t_actions     -90
# define   t_exit        -91
# define   t_adexpb      -92
# define   t_adexpu      -93
# define   t_collexp     -94
# define   t_list        -95
# define   t_dbset       -96
# define   t_predicate   -97
# define   t_ccexp       -98
# define   t_ccfac       -99
# define   t_ccterm      -100
# define   t_rangeb      -101
# define   t_ranger      -102
# define   t_rangel      -103
# define   t_rangei      -104
# define   t_next        -105
# define   c_sch         -1
# define   c_alg         -2
# define   c_rec         -3
# define   c_func        -4
# define   c_var         -5
# define   c_item        -6
# define   c_path        -7

int ref,e,lev_num,startd,starts,last,stack[100];
int nil -1;
int nul -1;

%}
%token T_ALGS T_ALG T_RECS T_REC T_ITEMS T_PATHS
%token T_FUNCS T_FUNC T_STRUCTURE T_VAR T_CONST T_BOOL
%token T_STR T_REAL T_INT T_END T_ARRAY T_OF T_BEGIN T_NOT
%token T_FOR T_DO T_ORDER T_ASCENDING T_DESCENDING T_MODIFY T_CREATE
%token T_DELETE T_ATTACH T_DETACH T_TRANSFER T_TO T_FROM T_VIA T_IF
%token T_THEN T_ELSE T_WHILE T_AND T_OR T_CALL T_INPUT T_PRINT
%token T_IDENTIFIER T_INT_K  T_REAL_K  T_STR_K T_BOOL_K T_OTHERS
%token T_NUL T_RELOP T_ARITHOP T_PAR_O T_PAR_C T_BRAK_O T_BRAK_C
%token T_ASSIGN T_COLON T_SEMICOLON T_COMMA T_PERIOD T_CROSS  T_EXCLA
%token T_PLUS T_MINUS T_MULT T_DIV T_EXP T_LIST T_SCHEMA T_ENDFOR
%token T_ENDIF  T_EXIT T_NEXT T_BRACEO T_BRACEC T_ENDWHILE
%start algorithm_structure
```

```
%% /* beginning  of  rule section.


                    Grammar.
                    ========


      1) General structure.
      ---------------------

*/

      algorithm_structure : algorithm_heading   algorithm_body
                {       LINK2($1,$2);
                        $$ = NODE(t_alg,$1);}

      algorithm_heading :  T_ALG  gname
                {       $$ = NODE(t_alh,$2);
                        INIT();  }


      gname :           T_IDENTIFIER
                {       e = FIND(nul,yytext,yyleng);
                        if ( e != -1)  ERROR(8);
                        else {
                        ref =CREATE(yytext,yyleng,c_alg,1,nil);
                        $$=TNODE(t_ident,ref);
                        CTREE(ref,$$);
                        }
                }

      algorithm_body :  extern_declaration_part
                         variable_declaration_part
                         statement_part
                {       LINK3($1,$2,$3);
                        $$ = NODE(t_alb, $1);    }
```

```
/*
        2) Declaration part.
        --------------------
*/

        extern_declaration_part : schema_declaration
                                  procedure_declaration
                                  function_declaration
                {       LINK3($1,$2,$3);
                        $$=NODE(t_exdp, $1); }


        schema_declaration :  T_SCHEMA sname
                {       $$ = NODE(t_schdec, $2);          }

                        |
                {       $$ = NODE(t_schdec, nil);         }

        sname :                 T_IDENTIFIER
                {       e = FIND(nul,yytext,yyleng);
                        if ( e != -1)  ERROR(1);
                        else {
                        ref=CREATE(yytext,yyleng,c_sch,1,nil);
                        $$=TNODE(t_ident,ref);
                        CTREE(ref,$$);
                        DECLARE();
                        }
                }

        procedure_declaration :    T_ALGS anames
                {       $$ = NODE(t_algdec, $2);          }

                        |
                {       $$ = NODE(t_algdec, nil);         }


        anames    :             aname_lst
                {       $$=NODE(t_names,$1); }

        aname_lst :             aname T_COMMA aname_lst
                {       $$ = $1;
                        LINK2($1,$3);    }

                        | aname
                {       $$=$1;           }

        aname :                 T_IDENTIFIER
                {       e = FIND(nul,yytext,yyleng);
                        if ( e != -1)  ERROR(2);
                        else {
                        ref=CREATE(yytext,yyleng,c_alg,1,nil);
                        $$=TNODE(t_ident,ref);
                        CTREE(ref,$$);
                        }
                }
```

```
function_declaration :    T_FUNCS name_list
        {       $$ = NODE(t_funcdec, $2);       }

                |
        {       $$ = NODE(t_funcdec, nil);      }

name_list :     fname T_COLON  variable_type T_COMMA name_list
        {       LINK2($1,$3);
                $$ =NODE(t_felt, $1);
                LINK2($$,$5);   }

                | fname T_COLON variable_type
        {       LINK2($1,$3);
                $$ =NODE(t_felt, $1);}

fname :         T_IDENTIFIER
        {       e = FIND(nul,yytext,yyleng);
                if ( e != -1)  ERROR(3);
                else {
                ref=CREATE(yytext,yyleng,c_func,1,nil);
                $$=TNODE(t_ident,ref);
                CTREE(ref,$$);
                }
        }
variable_declaration_part :
        {       $$ = NODE(t_vadp, nil); }

                | T_VAR variable_declarations
        {       $$ = NODE(t_vadp, $2);  }

variable_declarations : variable_declaration
                                variable_declarations
        {       $$ = $1;
                LINK2($1,$2);   }

                | variable_declaration
        {       $$= $1; }

variable_declaration : vnames T_COLON variable_type
        {       LINK2($1,$3);
                $$ = NODE(t_vardec, $1);        }

vnames   :      vname_lst
        {       $$=NODE(t_names,$1); }

vname_lst :     vname T_COMMA vname_lst
        {       $$ = $1;
                LINK2($1,$3);   }

                | vname
        {       $$=$1;          }

vname :         T_IDENTIFIER
        {       e = FIND(startd,yytext,yyleng);
                if ( e != -1)  ERROR(4);
                else {
                ref=CREATE(yytext,yyleng,c_var,lev_num,nil);
                $$=TNODE(t_ident,ref);
                CTREE(ref,$$);
                last=ref;
                }
        }
```

```
variable_type : simple
          {      $$ = $1;          }

                 | structured
          {      $$ = $1;          }

simple :         T_BOOL
          {      $$=NODE(t_bool, nil);    }

                 | T_STR
          {      $$=NODE(t_str, nil);     }

                 | T_REAL
          {      $$=NODE(t_real, nil);    }

                 | T_INT
          {      $$=NODE(t_int, nil);     }

structured :     structure
          {      $$=$1;            }

                 | array
          {      $$=$1;            }

                 | list_of
          {      $$=$1;            }

                 | db_rec
          {      $$=$1;            }

list_of :        T_LIST T_OF variable_type
          {      $$=NODE(t_listyp,$3); }

db_rec   :       T_REC rname
          {      $$=NODE(t_rectyp,$2); }

rname :          T_IDENTIFIER
          {      e = FIND(nul,yytext,yyleng);
                 if (e == -1) ERROR(5);
                 else if ( CLASS(e) != c_rec) ERROR(6);
                      else  $$=TNODE(t_ident,e);
          }

structure : T_STRUCTURE
          {      stack[lev_num] = startd;
                 startd = last;
                 lev_num++;        }

                 field_list T_END
          {      $$=NODE(t_struc, $3);
                 lev_num--;
                 startd = stack[lev_num]; }

field_list :     variable_declaration  field_list
          {      $$=$1;
                 LINK2($1,$2);     }

                 | variable_declaration
          {      $$=$1;            }
```

```
array :             T_ARRAY T_BRAK_O indexs T_BRAK_C T_OF
                    component_type
            {       LINK2($3, $6);
                    $$=NODE(t_array, $3); }
indexs   :          index_lst
            {       $$=NODE(t_indexs, $1);      }

index_lst :         index T_COMMA index_lst
            {       $$=$1;
                    LINK2($1, $3);              }

                    | index
            {       $$=$1;                      }

index :             T_INT_K
            {       $$=TNODE(t_int_k, PUTK(yytext, yyleng, 2)); }

component_type :  simple
            {       $$=$1;                      }

                    | structure
            {       $$=$1;                      }

                    | list_of
            {       $$=$1;              }

                    | db_rec
            {       $$=$1;              }
```

```
/*
        3) Statement part.
        ------------------
*/
        statement_part : T_BEGIN statements T_END
                 {       $$ = NODE(t_stp, $2); }


        statements :     statement  T_SEMICOLON  statements
                 {       $$ = $1;
                         LINK2($1,$3);     }

                         | statement
                 {       $$ = $1;          }

        statement :      ass_st
                 {       $$=$1;            }

                         | for_st
                 {       $$=$1;            }

                         | while_st
                 {       $$=$1;            }

                         | db_mod
                 {       $$=$1;            }

                         | call_st
                 {       $$=$1;            }

                         | cond_st
                 {       $$=$1;            }

                         | next_st
                 {       $$=$1;            }

                         | exit_st
                 {       $$=$1;            }

                         | dummy
                 {       $$=$1;            }

dummy                :
                 {       $$=nil;           }
```

```
/*

        7) Enumerative loop.
        --------------------

*/
        for_st    :        T_FOR variable T_ASSIGN for_list

                           T_DO actions T_ENDFOR
                  {        LINK3($2, $4, $6);
                           $$=NODE(t_enumloop, $2);   }


        for_list :         general_expression ord
                  {        LINK2($1, $2);
                           $$=NODE(t_forlis, $1); }

                           | general_expression
                  {        $$=NODE(t_forlis, $1);    }

        ord            :   T_ORDER order_keys
                  {        $$=NODE(t_order, $2);     }

        order_keys   :     name  ad T_COMMA order_keys
                  {        LINK2($1, $2);
                           $$=NODE(t_ordkey, $1);
                           LINK2($$, $4);            }

                           | name ad
                  {        $$=NODE(t_ordkey, $1);
                           LINK2($1, $2);    }


        ad             :   T_ASCENDING
                  {        $$=NODE(t_asc, nil);      }

                           | T_DESCENDING
                  {        $$=NODE(t_des, nil);      }

                           |
                  {        $$=NODE(t_nad, nil);      }
```

```
/*
        8) Data base modification.
        --------------------------
*/
        db_mod :        modif
                  {     $$=NODE(t_dbmod,$1);      }

                        | creat
                  {     $$=NODE(t_dbmod,$1);      }

                        | del
                  {     $$=NODE(t_dbmod,$1);      }

                        | att
                  {     $$=NODE(t_dbmod,$1);      }

                        | det
                  {     $$=NODE(t_dbmod,5,$1);    }

                        | transf
                  {     $$=NODE(t_dbmod,6,$1);    }

        modif  :        T_MODIFY name  belonging_condition
                  {     LINK2($2,$3);
                        $$=NODE(t_modify,$2);     }

        creat  :        T_CREATE name T_ASSIGN name  belonging_condition
                  {     LINK3($2,$4,$5);
                        $$=NODE(t_creat,$2);      }

        del    :        T_DELETE name
                  {     $$=NODE(t_del,$2);        }

        att    :        T_ATTACH name T_TO name T_VIA name
                  {     LINK3($2,$4,$6);
                        $$=NODE(t_att,$2);        }

        det    :        T_DETACH name T_FROM name T_VIA name
                  {     LINK3($2,$4,$6);
                        $$=NODE(t_det,$2);        }

                        | T_DETACH name T_VIA name
                  {     LINK2($2,$4);
                        $$=NODE(t_det,$2);        }

        transf :        T_TRANSFER name T_FROM name

                        T_TO name T_VIA name
                  {     LINK4($2,$4,$6,$8);
                        $$=NODE(t_transf,$2);     }

                        | T_TRANSFER name T_TO name T_VIA name
                  {     LINK3($2,$4,$6);
                        $$=NODE(t_transf,$2);     }
```

```
/*

        9) Alternative.
        ---------------
*/
        cond_st :          T_IF general_expression T_THEN actions  T_ENDIF
                    {      LINK2($2,$4);
                           $$=NODE(t_altern,$2);    }

                           | T_IF general_expression T_THEN actions

                             T_ELSE actions T_ENDIF
                    {      LINK3($2,$4,$6);
                           $$=NODE(t_altern,$2);    }


        actions :          statements
                    {      $$=NODE(t_actions,$1);           }

/*

        10) While loop.
        ---------------
*/
        while_st :         T_WHILE general_expression T_DO actions T_ENDWHILE
                    {      LINK2($2,$4);
                           $$=NODE(t_whileloop,$2);         }
/*
        12) Call statement.
        -------------------
*/
        call_st    :       name parameter_part
                    {      LINK2($1,$2);
                           $$=NODE(t_call,$1);      }
/*
        13) Let statement.
        ------------------
*/
        ass_st :           variable T_ASSIGN general_expression
                    {      LINK2($1,$3);
                           $$=NODE(t_assst,$1);     }

/*

        14) Next statement.
        ----------------------
*/
        next_st :          T_NEXT name
                    {      $$=NODE(t_next,$2);      }

                           | T_NEXT
                    {      $$=NODE(t_next,nil);     }
```

```
/*

        15) Exit statement.
        ----------------------

*/

        exit_st :           T_EXIT name
                    {       $$=NODE(t_exit,$2);         }

                            | T_EXIT
                    {       $$=NODE(t_exit,nil);        }

/*

        16) Function designation.
        -------------------------

*/

        func      :         name parameter_part
                    {       LINK2($1,$2);
                            $$=NODE(t_func,$1);         }


        parameter_part : T_PAR_O  parameter_list T_PAR_C
                    {       $$=NODE(t_parlis,$2);       }

                            | T_PAR_O T_PAR_C
                    {       $$=NODE(t_parlis,nil);      }

        parameter_list : general_expression  T_COMMA parameter_list
                    {       LINK2($1,$3);               }

                            | general_expression
                    {       $$=$1;                      }
```

```
/*      17) Expression.
        ---------------
*/

        general_expression : boolean_expression
                {       $$=NODE(t_genexp,$1);    }

        boolean_expression : boolean_factor T_OR boolean_expression
                {       LINK2($1,$3);
                        $$=NODE(t_orexp,$1);     }

                        | boolean_factor
                {       $$=$1;          }

        boolean_factor : boolean_term T_AND boolean_factor
                {       LINK2($1,$3);
                        $$=NODE(t_andexp,$1);    }

                        | boolean_term
                {       $$=$1;          }


        boolean_term :  T_NOT boolean_primary
                {       $$=NODE(t_not,$2);       }

                        | boolean_primary
                {       $$=$1;          }

        boolean_primary : logical_value
                {       $$=$1;          }

                        | test_expression
                {       $$=$1;          }

                        | arithmetic_expression
                {       $$=$1;          }

        test_expression : arithmetic_expression  test_op
                          arithmetic_expression
                {       LINK3($2,$1,$3);
                        $$=NODE(t_tesexp, $2);   }

        arithmetic_expression : adding_expression

                {       $$=NODE(t_arexp,$1);     }

        adding_expression :  factor adding arithmetic_expression
                {       LINK3($2,$1,$3);
                        $$=NODE(t_adexpb,$2);    }

                        | adding  factor
                {       LINK2($1,$2);
                        $$=NODE(t_adexpu,$1);    }

                        | factor
                {       $$=$1;          }
```

```
factor :        term  multiplying  factor
          {     LINK3($2,$1,$3);
                $$=NODE(t_mulexp,$2);    }

                | term
          {     $$=$1;          }

term :          primary  T_EXP term
          {     LINK2($1,$3);
                $$=NODE(t_exexp,$1);     }

                | primary
          {     $$=$1;          }


primary :       str_k
          {     $$=$1;   }

                | unsigned_number
          {     $$=$1;          }

                | collection_expression
          {     $$=$1;          }

                | func
          {     $$=$1;          }

                | variable
          {     $$=$1;          }

                | T_PAR_O arithmetic_expression T_PAR_C
          {     $$=$2;          }


adding :        T_PLUS
          {     $$=TNODE(t_plus,nil);}

                | T_MINUS
          {     $$=TNODE(t_minus,nil);}

multiplying :   T_MULT
          {     $$=NODE(t_mult,nil);     }

                | T_DIV
          {     $$=NODE(t_div,nil);      }
```

```
/*
        18) Collection expression.
        ---------------------------
*/

collection_expression : range
            {       $$=NODE(t_collexp,$1);   }

                    | list
            {       $$=NODE(t_collexp,$1);   }

                    | db_object_set
            {       $$=NODE(t_collexp,$1);   }

list  :             T_BRACEO list_enumeration T_BRACEC
            {       $$=NODE(t_list,$2);      }

list_enumeration : list_element T_COMMA list_enumeration
            {       LINK2($1,$3);
                    $$=$1;           }

                    | list_element
            {       $$=$1;           }

list_element :      general_expression
            {       $$=$1;           }

db_object_set : name  predicate
            {       LINK2($1,$2);
                    $$=NODE(t_dbset,$1);     }

predicate     :     coll_cond_exp
            {       $$=NODE(t_predicate, $1);        }

coll_cond_exp : coll_cond_factor T_OR  coll_cond_exp
            {       LINK2($1,$3);
                    $$=NODE(t_ccexp, $1);    }

                    | coll_cond_factor
            {       $$=$1;           }


coll_cond_factor :  coll_cond_term T_AND coll_cond_factor
            {       LINK2($1,$3);
                    $$=NODE(t_ccfac, $1);    }

                    | coll_cond_term
            {       $$=$1;           }

coll_cond_term :    T_NOT coll_cond_primary
            {       $$=NODE(t_ccterm, $2);   }

                    | coll_cond_primary
            {       $$=$1;                   }
```

```
coll_cond_primary :  relation_condition
        {       $$=$1;                          }

                | belonging_condition
        {       $$=$1;                          }

                | T_PAR_O  predicate  T_PAR_C
        {       $$=$2;                          }


relation_condition :  relation_operator  db_object_set T_PAR_C
        {       LINK2($1,$2);
                $$=NODE(t_relcon,$1);   }

                     | relation_operator  variable  T_PAR_C
        {       LINK2($1,$2);
                $$=NODE(t_relcon,$1);   }

relation_operator :  T_PAR_O name T_COLON co
        {       LINK2($4,$2);
                $$=NODE(t_relop,$4);    }

                | T_PAR_O T_COLON co
        {       $$=NODE(t_relop,$3);    }

belonging_condition :    test_op  arithmetic_expression
        {       LINK2($1,$2);
                $$=NODE(t_belcon,$1);   }
```

```
/*
        19) Variable.
        -------------
*/
variable :                simple_variable
                    {     $$=NODE(t_var, $1);        }

                          | subscripted_variable
                    {     $$=NODE(t_var, $1);        }

                          | structured_variable
                    {     $$=NODE(t_var, $1);        }

simple_variable :         name
                    {     $$=$1;            }

subscripted_variable :    name T_BRAK_O subscripts T_BRAK_C
                    {     LINK2($1, $3);
                          $$=NODE(t_arrelt, $1);     }

subscripts :              subscript_list
                    {     $$=NODE(t_subls, $1); }

subscript_list :          general_expression T_COMMA  subscript_list
                    {     LINK2($1, $3);             }

                          | general_expression
                    {     $$=$1;                     }

structured_variable :     name T_PERIOD
                     {    starts = SON($1);          }

                          substructure
                    {     LINK2($1, $4);
                          $$=NODE(t_struct, $1);
                          starts = nul;              }

substructure :            name T_PERIOD
                    {     starts = SON($1);          }

                          substructure
                    {     LINK2($1, $4);             }

                          | name
                    {     $$=$1;                     }

name :                    T_IDENTIFIER
                    {     e = FIND(starts, yytext, yyleng);
                          if ( e == nul) ERROR(7);
                          else  $$=TNODE(t_ident, e);
                    }
```

```
unsigned_number :           T_INT_K
                   {        $$=TNODE(t_unsnum,PUTK(yytext,yyleng));}

                            | T_REAL_K
                   {        $$=TNODE(t_unsnum,PUTK(yytext,yyleng));}

str_k     :                 T_STR_K
                   {        $$=TNODE(t_str_k,PUTK(yytext,yyleng));}

logical_value   :           T_BOOL_K
                   {        $$=TNODE(t_logval,PUTK(yytext,yyleng));}


co              :           cardinal
                   {        $$=$1;}

                            | ordinal
                   {        $$=$1;}
                            |
                   {        $$=NODE(t_co,nil);}

test_op         :           T_RELOP
                   {        $$=TNODE(t_tesop,PUTK(yytext,yyleng));}

ordinal :                   T_CROSS general_expression
                   {        $$=NODE(t_ord,$2);         }

cardinal :                  T_MULT
                   {        $$=NODE(t_card, nil);    }

                            | range
                   {        $$=NODE(t_card,$1);        }

range :                     T_BRAK_O general_expression T_COLON
                                     general_expression T_BRAK_C
                   {        LINK2($2,$4);
                            $$=NODE(t_rangeb,$2);    }


                             | T_BRAK_O  T_MULT  T_COLON
                                     general_expression T_BRAK_C
                   {        $$=NODE(t_ranger,$4);    }

                             | T_BRAK_O general_expression T_COLON
                                     T_MULT  T_BRAK_C
                   {        $$=NODE(t_rangel,$2);    }

                             | T_BRAK_O T_MULT T_COLON T_MULT T_BRAK_C
                   {        $$=NODE(t_rangei, nil); }
```

```
%%
# include "lex.yy.c"
# include "newadl.c"
# include "indent.c"
main(){
        char text[256];
        int i,j;
        i=yyparse();
                fprintf(yyout,"%s \t %s\t %s\t %s \t %s \t %s\n",
                "line number","ident","type","f_son","brother","father");
                for ( i = 0; i <= new; i++) {

                        fprintf(yyout,"%8s %3d\t","ligne : ",i);
                        fprintf(yyout,"%3d \t",nde[i].identifier);
                        fprintf(yyout,"%3d \t",nde[i].type);
                        fprintf(yyout,"%3d \t",nde[i].f_son);
                        fprintf(yyout,"%3d \t",nde[i].brother);
                        fprintf(yyout,"%3d \n",nde[i].father);
                }
                fprintf(yyout,"%s \t %s\t %s\t %s \t %s \t %s\n",
                "line number","length","class","level","ref to tree","value"
                for (i=0; i < fnempty ; i++) {
                        fprintf(yyout,"\n %8s %3d \t","ligne : ",i);
                        fprintf(yyout,"%3d \t",symb[i].leng);
                        fprintf(yyout,"%3d \t",symb[i].class);
                        fprintf(yyout,"%3d \t",symb[i].lev);
                        fprintf(yyout,"%3d \t",symb[i].tree);
                        GET_NAME(i,text);
                        fprintf(yyout,text);
                        fprintf(yyout,"\n");
                }
                fprintf(yyout,"%s \t %s\t %s\n",
                        "line number","length","value");
                for ( i=0; i < fkempty; i++) {
                        fprintf(yyout,"\n %8s %3d \t","ligne : ",i);
                        fprintf(yyout,"%3d \t",cons[i].lengk);
                        GETK(i,text);
                        fprintf(yyout,text);
                        fprintf(yyout,"\n");
                }
        INDENT(new);
        }

yyerror(s) char *s; {
        fprintf(yyout,"%s \n",s);
        return(1);
}

INIT()

{
        startd=nul;
        starts=nul;
        last=nul;
        lev_num=1;
}
```

```
DECLARE()

{
        CREATE("CUSTOMER",8,c_rec,1,nil);
        CREATE("ORDER",5,c_rec,1,nil);
        CREATE("ORDER_LINE",10,c_rec,1,nil);
        CREATE("PRODUCT",7,c_rec,1,nil);
        CREATE("NAME",4,c_item,1,nil);
        CREATE("NUM",3,c_item,1,nil);
        CREATE("QUANTITY",8,c_item,1,nil);
        CREATE("CUSOR",5,c_path,1,nil);
        CREATE("ORCUS",5,c_path,1,nil);
        CREATE("ORLI",4,c_path,1,nil);
        CREATE("LIOR",4,c_path,1,nil);
        CREATE("LIPRO",5,c_path,1,nil);
        CREATE("PROLI",5,c_path,1,nil);
        return(0);
}

ERROR(i)

int i;

{
        switch (i) {

        case 1 :
fprintf(yyout,"%s %s %s \n","schema ",yytext," is multiply defined.");
                    break;

        case 2 :
fprintf(yyout,"%s %s %s \n","algorithm",yytext," is multiply defined");
                    break;

        case 3 :
fprintf(yyout,"%s %s %s \n","function ",yytext," is multiply defined");
                    break;

        case 4 :
fprintf(yyout,"%s %s %s \n","variable ",yytext," is multiply defined");
                    break;

        case 5 :
fprintf(yyout,"%s %s %s \n","record ",yytext," is undefined");
                    break;

        case 6 :
fprintf(yyout,"%s %s %s \n","name ",yytext," should be a record type");
                    break;

        case 7 :
fprintf(yyout,"%s %s %s \n","name ",yytext," is undefined");
                    break;

        case 8 :
fprintf(yyout,"%s %s %s \n","algorithm ",yytext," is multiply defined");
                    break;
        }
}
```

```
/*

        a) General declarations.
        .........................
*/

int     false 0;
int      true 1;

/*      1) Node manipulations routines.
        ------------------------------

        a) General declarations.
        .......................

        Node structure definition :
                1)Identifier : used to identifie a node.
                2)Type        : indicates the type of the node (see BNF def.)
                3)F_son      : reference to the first son if not NIL.
                4)Brother     : reference to the brother if not NIL.
                5)Father      : reference to the father if not NIL.

*/

int     id  0;
int     new -1;
struct  {
                int     identifier;
                int     type;
                int     f_son;
                int     brother;
                int     father; } nde[1000];

/*      b) Node creation.
        ..................
        Input  : nt => node type.
                 fs => value of first son (NIL if any).

        Output : reference to the node.
*/

        NODE(nt,fs)

int             nt,fs;

{
                int cur
                new++;
                nde[new].identifier = ++id;
                nde[new].type        = nt;
                nde[new].brother     = nil;
                nde[new].father      = nil;
                nde[new].f_son      = fs;

                cur = fs;
                while  ( cur != nil ) {
                                nde[cur].father = new;
                                cur = nde[cur].brother;
                }

                return(new);
}
```

```
/*      b) Terminal node creation.
        ..............................
        The difference with respect to NODE is that TNODE does not
        put the reference of the node in all the son. TNODE is
        usually used when defining a terminal node. In that case
        'fs' is different from nil but does not indicate a node
        of the tree.( in that particular case, 'fs' is the refrence
        of an entry in the symbol table.

        Input  : nt => node type.
                 fs => value of first son (NIL if any).

        Output : reference to the node.
*/

        TNODE(nt,fs)

int             nt,fs;

{
                new++;
                nde[new].identifier = ++id;
                nde[new].type        = nt;
                nde[new].brother     = nil;
                nde[new].f_son       = fs;

                return(new);
}



/*
        C) Link  2  brothers.
        .......................

        Input  : s1 => reference to the first brother.
                 s2 => reference to the second brother.
*/

        LINK2(s1,s2)

        int  s1,s2;

{
                nde[s1].brother = s2;
                return(0);
}
```

```
/*
        d) Link  3  brothers.
        ......................

        Input  : s1 => reference to the first brother.
                 s2 => reference to the second brother.
                 s3 => reference to the third brother.
*/

        LINK3(s1, s2, s3)

        int  s1, s2, s3;

{
                nde[s1].brother = s2;
                nde[s2].brother = s3;
                return(0);
}
/*
        e) Link  4  brothers.
        ......................

        Input  : s1 => reference to the first brother.
                 s2 => reference to the second brother.
                 s3 => reference to the third brother.
                 s4 => reference to the fourth brother.
*/

        LINK4(s1, s2, s3, s4)

        int  s1, s2, s3, s4;

{
                nde[s1].brother = s2;
                nde[s2].brother = s3;
                nde[s3].brother = s4;
                return(0);
}
/*
        f) Link  5  brothers.
        ......................

        Input  : s1 => reference to the first brother.
                 s2 => reference to the second brother.
                 s3 => reference to the third brother.
                 s4 => reference to the fourth brother.
                 s5 => reference to the fifth brother.
*/

        LINK5(s1, s2, s3, s4, s5)

        int  s1, s2, s3, s4, s5;

{
                nde[s1].brother = s2;
                nde[s2].brother = s3;
                nde[s3].brother = s4;
                nde[s4].brother = s5;
                return(0);
}
```

```
/*      g) Access to the field 'ident'.
        ..............................
*/
        IDENT(r)

int     r;

{
        if (r > new) fprintf(yyout," ref OUT OF RANGE (func.IDENT)");
        return(nde[r].identifier);
}

/*      h) Access to the field 'type'.
        ..............................
*/
        TYPE(r)

int     r;

{
        if (r > new) fprintf(yyout," ref OUT OF RANGE (func.TYPE)");
        return(nde[r].type);
}

/*      i) Access to the field 'father'.
        ................................
*/
        FATHER(r)

int     r;

{
        if (r > new) fprintf(yyout," ref OUT OF RANGE (func.FATHER)");
        return(nde[r].father);
}

/*      j) Access to the field 'brother'.
        .................................
*/
        BROTHER(r)

int     r;

{
        if (r > new) fprintf(yyout," ref OUT OF RANGE (func.BROTHER)");
        return(nde[r].brother);
}

/*      k) Access to the field 'son'.
        .............................
*/
        SON(r)

int     r;

{
        if (r > new) fprintf(yyout," ref OUT OF RANGE (func.SON)");
        return(nde[r].f_son);
}
```

```
/*      l)Update the field 'ident'.
        ...........................
*/
        CIDENT(r,v)

int     r,v;

{
        if (r > new) fprintf(yyout," ref OUT OF RANGE (func.IDENT)");
        nde[r].identifier = v;
        return(0);
}
/*      m) Update the field 'type'.
        ...........................
*/
        CTYPE(r,v)

int     r,v;

{
        if (r > new) fprintf(yyout," ref OUT OF RANGE (func.TYPE)");
        nde[r].type = v;
        return(0);
}

/*      n)Update the field 'father'.
        ............................
*/
        CFATHER(r,v)

int     r,v;

{
        if (r > new) fprintf(yyout," ref OUT OF RANGE (func.FATHER)");
        nde[r].father = v;
        return(0);
}
/*      n)Update the field 'brother'.
        .............................
*/
        CBROTHER(r,v)

int     r,v;

{
        if (r > new) fprintf(yyout," ref OUT OF RANGE (func.BROTHER)");
        nde[r].brother = v;
        return(0);
}
/*      o) Update the field 'son'.
        ..........................
*/
        CSON(r,v)

int     r,v;

{
        if (r > new) fprintf(yyout," ref OUT OF RANGE (func.SON)");
        nde[r].f_son = v;
        return(0);
}
```

```
/*      2) Symbol table manipulation routines.
        --------------------------------------
*/

int     fnempty 0;
struct{
                int     lev;
                int     tree;
                int     leng;
                int     class;
                int     val ;  } symb[200];
int     cur_name 0;
char    tabname[2000];
```

```
/*      a) Find a string in the symbol table
        ....................................

        Input  : ref    => contains the reference of the strating entry.
                 text   => contains the string.
                 length => indicates the length of the string.

        Output : the type of the string in the symbol table.
*/
        FIND( ref, text , length )

char    text[];
int     length,ref;
{
                int     cur;
                int     found;
                int     i,ii;
                int     level_s,level_f;

                found = 0;
                if (ref == -1 ) {
                        level_s = 0;
                        level_f = 1;
                        cur     = 0;
                }
                else {
                        level_s = symb[ref].lev;
                        level_f = level_s+1;
                        cur = ref+1;
                }
                while ((     cur < fnempty)
                && (found == false)
                && (symb[cur].lev > level_s)) {

                        if    ((symb[cur].leng == length)
                           && (symb[cur].lev == level_f))  {

                                i = 0;
                                ii= symb[cur].val;
                                found = true;
                                while  ((i < length) && (found == true)) {
                                        if (tabname[ii] != text[i]) {
                                                found = false;
                                        }
                                        i++;
                                        ii++;
                                }
                        }
                        cur++;
                }
/*              fprintf(yyout,"%s %1d \n","found = ",found);*/
                if  (found == false)    return(-1);
                else                    return(--cur);
}
```

```
/*      b) Creation of an entry in the symbol table.
        .........................................

        Input  : text   => contains the string.
                 length => indicates the length of the string.
                 class   => indicates the class of the string.

        Rem : -tabname is a table of characters used to store the names.

              -the field 'value' contains the index value of the first
               character of the name corresponding to the entry.
*/

        CREATE( text, length, class,level,ref)

char    text[];
int     length, class,level,ref;

{
                int     i,cur;

                cur=fnempty++;
                if ( length > 16 ) length = 16;
                symb[cur].leng = length;
                symb[cur].class= class;
                symb[cur].lev  = level;
                symb[cur].tree = ref;
                symb[cur].val  = cur_name;
                for(i=0; i<length ; i++) {
                         tabname[cur_name]=text[i];
                         cur_name++;
                }
                return(cur);
}
/*      c) Access to the name of an entry.
        ..................................
*/

        GET_NAME(i,txt)

        char txt[];
        int i;

{
        int cur,lim,base;

        if ( i >= fnempty ) fprintf(yyout,"out of range (func.GET_NAME)");
        lim = symb[i].leng;
        base = symb[i].val;
        for ( cur = 0 ;cur < lim ; txt[cur] = tabname[base+cur++]);
        txt[symb[i].leng]='\0';
        return(symb[i].leng);
}
```

```
/*      d) Access to the field 'tree'.
        ..............................
*/
        TREE(r)

int     r;

{
        if (r >= fnempty) fprintf(yyout,"ref OUT OF RANGE (func.TREE)");
        return(symb[r].tree);
}

/*      e) Update the field 'tree'.
        ...........................
*/
        CTREE(r,v)

int     r,v;

{
        if (r >= fnempty) fprintf(yyout,"ref OUT OF RANGE (func.CTREE)");
        symb[r].tree = v;
        return(0);
}

/*      f) Access to the field 'class'.
        ...............................
*/
        CLASS(r)

int     r;

{
        if (r >= fnempty) fprintf(yyout,"ref OUT OF RANGE (func.CLASS)");
        return(symb[r].class);
}

/*      g) Update the field 'class'.
        ............................
*/
        CCLASS(r,v)

int     r,v;

{
        if (r >= fnempty) fprintf(yyout,"ref OUT OF RANGE (func.CCLASS)");
        symb[r].class = v;
        return(0);
}
```

```
/*      3) Constant table manipulation routines.
        ---------------------------------------

*/

int     cur_cons 0;
int     fkempty;
struct  {
                int     lengk;
                int     valk;   } cons[200];
char    tabcons[2000];


/*      a) Constant creation.
        . . . . . . . . . . . . . . . . . . . .

*/
        PUTK(text,length)

        char text[];
int  length;

{
                int     i,cur;

                cur=fkempty++;
                cons[cur].lengk = length;
                cons[cur].valk  = cur_cons;
                for(i=0; i<length ; i++) {
                         tabcons[cur_cons]=text[i];
                         cur_cons++;
                }
                return(cur);
}

/*      b) Access a constant.
        . . . . . . . . . . . . . . . . . . . .

*/

        GETK(i,txt)

        char txt[];
        int i;

{

        int cur,lim,base;


        if ( i >= fkempty ) fprintf(yyout,"out of range (func.GETK)");
        lim = cons[i].lengk;
        base = cons[i].valk;
        for ( cur = 0 ;cur < lim ; txt[cur] = tabcons[base+cur++]);
        txt[cons[i].lengk]='\0';
        return(cons[i].lengk);
}
```

```
/* REPRESENTATION DE L'ARBRE SYNTAXIQUE PAR INDENTATION */

int ind_profondeur 0;

INDENT(ref)
int ref;
        {

        char txt[256];

        if (ref != nil){
          pr_blancs(ind_profondeur);
          fprintf(yyout,"%s %d ",c_type(nde[ref].type),
                                      nde[ref].identifier);
          fprintf(yyout,"%s %d \n",c_type(nde[nde[ref].father].type),
                                   nde[nde[ref].father].identifier);

          ind_profondeur++;

          if(nde[ref].type == t_unsnum ||
             nde[ref].type == t_int_k ||
             nde[ref].type == t_logval ||
             nde[ref].type == t_str_k ||
             nde[ref].type == t_tesop ){

             pr_blancs(ind_profondeur);
             GETK(nde[ref].f_son,txt);
             fprintf(yyout,"%s %d ",txt,nde[ref].f_son);
             fprintf(yyout,"%s %d\n",c_type(nde[ref].type),
                                      nde[ref].identifier);
              }
          else if (nde[ref].type == t_ident) {
                  pr_blancs(ind_profondeur);
                  GET_NAME(nde[ref].f_son,txt);
                  fprintf(yyout,"%s %d ",txt,
                                      nde[ref].f_son);
                  fprintf(yyout,"%s %d\n",c_type(nde[ref].type),
                                      nde[ref].identifier);

                }
                else     INDENT(nde[ref].f_son);
          ind_profondeur--;
          INDENT(nde[ref].brother);
        }

} /* end INDENT */


/* impression de blancs */

pr_blancs(longueur)
int longueur;

{ int i;
        for(i=0;i<longueur;++i) fprintf(yyout,"%s","! ");

} /* end pr_blancs */
```

```
/* conversion du type d' un noeud */

char     *c_type(i)

         int i;

{
         switch (i) {

                   case t_int_k     : return("t_int_k"); /* refer to the */
                   case t_tesop     : return("t_tesop"); /* symbol table  */
                   case t_unsnum    : return("t_unsnum");
                   case t_ident     : return("t_ident");
                   case t_str_k     : return("t_str_k");
                   case t_logval    : return("t_logval");

                   case t_alg       : return("t_alg"); /* declarations */
                   case t_alh       : return("t_alh");
                   case t_alb       : return("t_alb");
                   case t_exdp      : return("t_exdp");
                   case t_algdec    : return("t_algdec");
                   case t_funcdec   : return("t_funcdec");
                   case t_felt      : return("t_felt");
                   case t_vadp      : return("t_vadp");
                   case t_vardec    : return("t_vardec");
                   case t_bool      : return("t_bool");
                   case t_real      : return("t_real");
                   case t_int       : return("t_int");
                   case t_rectyp    : return("t_rectyp");
                   case t_array     : return("t_array");
                   case t_struc     : return("t_struc");
                   case t_names     : return("t_names");
                   case t_indexs    : return("t_indexs");
                   case t_schdec    : return("t_schdec");
                   case t_listyp    : return("t_listyp");
                   case t_str       : return("t_str");

                   case t_stp       : return("t_stp"); /* statement part */
                   case t_enumloop  : return("t_enumloop");
                   case t_forlis    : return("t_forlis");
                   case t_order     : return("t_order");
                   case t_ordkey    : return("t_ordkey");
                   case t_asc       : return("t_asc");
                   case t_des       : return("t_des");
                   case t_nad       : return("t_nad");
                   case t_dbmod     : return("t_dbmod");
                   case t_modify    : return("t_modify");
                   case t_creat     : return("t_creat");
                   case t_del       : return("t_del");
                   case t_att       : return("t_att");
                   case t_det       : return("t_det");
                   case t_transf    : return("t_transf");
                   case t_altern    : return("t_altern");
                   case t_whileloop : return("t_whileloop");
                   case t_call      : return("t_call");
                   case t_assst     : return("t_assst");
                   case t_func      : return("t_func");
                   case t_next      : return("t_next");
                   case t_parlis    : return("t_parlis");
                   case t_actions   : return("t_actions");
                   case t_exit      : return("t_exit");
```

```
            case t_genexp   : return("t_genexp");   /* expression */
            case t_orexp    : return("t_orexp");
            case t_andexp   : return("t_andexp");
            case t_not      : return("t_not");
            case t_tesexp   : return("t_tesexp");
            case t_arexp    : return("t_arexp");
            case t_mulexp   : return("t_mulexp");
            case t_exexp    : return("t_exexp");
            case t_plus     : return("t_plus");
            case t_minus    : return("t_minus");
            case t_mult     : return("t_mult");
            case t_div      : return("t_div");
            case t_relop    : return("t_relop");
            case t_belcon   : return("t_belcon");
            case t_var      : return("t_var");
            case t_arrelt   : return("t_arrelt");
            case t_struct   : return("t_struct");
            case t_co       : return("t_co");
            case t_ord      : return("t_ord");
            case t_card     : return("t_card");
            case t_relcon   : return("t_relcon");
            case t_subls    : return("t_subls");
            case t_adexpb   : return("t_adexpb");
            case t_adexpu   : return("t_adexpu");
            case t_collexp  : return("t_collexp");
            case t_list     : return("t_list");
            case t_dbset    : return("t_dbset");
            case t_predicate        : return("t_predicate");
            case t_ccexp    : return("t_ccexp");
            case t_ccfac    : return("t_ccfac");
            case t_ccterm   : return("t_ccterm");
            case t_rangeb   : return("t_rangeb");
            case t_ranger   : return("t_ranger");
            case t_rangel   : return("t_rangel");

            default         : return("UNKNOWN");
                    }

} /* end c_type */
```

Annexe 4 : Examples of parse trees.

This annexe contains six examples which show the work done by the adl parser. Each example shows :

- the text of the ADL program.

- the table that contains the nodes of the parse tree corresponding to the ADL program.

- the symbol table associated to the parse tree.

- the constant table associated to the parse tree.

- a "readable" view of the parse tree. The character ´!´ is used to shows the depth of the node in the tree. Each line represents either a node, or the value of a name, or the value of a constant. If the first field (after the ´!´) denotes a node type, then the line represents a node, otherwise it represents either a constant, or a name. If a node "N" is represented by the line ´n´, the first son of "N" is represented by the line ´n´ + 1.

  If the line represents a node, then the first field gives the type of the node, the second field gives the "identifier" value of the node, the third field gives the type of the node that is the father of the current node, finally, the fourth field gives the "identifier" value of the node that is the father of the current node.

  If the line represents the value of a constant or a name,then the first field gives the value of the constant -or the name, the second field gives the reference of the constant -or the name, in the table of constant -or the symbol table.

```
algorithm  alg1
funcs sin:real
var a,b,c : real
    d,e,f : integer
    g,h,i : string

/* this is a comment */

begin
        a:= 33.2 - sin(447.);
        d:= 34 + 8;
        h:= 'hello the world';
end
```

1) The node table.

	line number	ident	type	f_son	brother	father
ligne :	0	1	-18	0	-1	1
ligne :	1	2	-6	0	60	61
ligne :	2	3	-88	-1	3	8
ligne :	3	4	-11	-1	7	8
ligne :	4	5	-18	1	5	6
ligne :	5	6	-21	-1	-1	6
ligne :	6	7	-13	4	-1	7
ligne :	7	8	-12	6	-1	8
ligne :	8	9	-7	2	27	60
ligne :	9	10	-18	2	10	12
ligne :	10	11	-18	3	11	12
ligne :	11	12	-18	4	-1	12
ligne :	12	13	-27	9	13	14
ligne :	13	14	-21	-1	-1	14
ligne :	14	15	-17	12	20	27
ligne :	15	16	-18	5	16	18
ligne :	16	17	-18	6	17	18
ligne :	17	18	-18	7	-1	18
ligne :	18	19	-27	15	19	20
ligne :	19	20	-22	-1	-1	20
ligne :	20	21	-17	18	26	27
ligne :	21	22	-18	8	22	24
ligne :	22	23	-18	9	23	24
ligne :	23	24	-18	10	-1	24
ligne :	24	25	-27	21	25	26
ligne :	25	26	-20	-1	-1	26
ligne :	26	27	-17	24	-1	27
ligne :	27	28	-16	14	59	60
ligne :	28	29	-18	2	-1	29
ligne :	29	30	-51	28	41	42

line number		ident	type	f_son	brother	father
ligne :	30	31	-54	0	38	39
ligne :	31	32	-45	-1	30	39
ligne :	32	33	-18	1	36	37
ligne :	33	34	-54	1	-1	34
ligne :	34	35	-40	33	-1	35
ligne :	35	36	-35	34	-1	36
ligne :	36	37	-87	35	-1	37
ligne :	37	38	-86	32	-1	38
ligne :	38	39	-40	37	-1	39
ligne :	39	40	-92	31	-1	40
ligne :	40	41	-40	39	-1	41
ligne :	41	42	-35	40	-1	42
ligne :	42	43	-83	29	52	59
ligne :	43	44	-18	5	-1	44
ligne :	44	45	-51	43	51	52
ligne :	45	46	-54	2	48	49
ligne :	46	47	-44	-1	45	49
ligne :	47	48	-54	3	-1	48
ligne :	48	49	-40	47	-1	49
ligne :	49	50	-92	46	-1	50
ligne :	50	51	-40	49	-1	51
ligne :	51	52	-35	50	-1	52
ligne :	52	53	-83	44	58	59
ligne :	53	54	-18	9	-1	54
ligne :	54	55	-51	53	57	58
ligne :	55	56	-20	4	-1	56
ligne :	56	57	-40	55	-1	57
ligne :	57	58	-35	56	-1	58
ligne :	58	59	-83	54	-1	59
ligne :	59	60	-65	42	-1	60
ligne :	60	61	-25	8	-1	61
ligne :	61	62	-5	1	-1	-1

2) The symbol table.

line number		length	class	level	tree	value
ligne :	0	4	-2	1	0	alg1
ligne :	1	3	-4	1	4	sin
ligne :	2	1	-5	1	9	a
ligne :	3	1	-5	1	10	b
ligne :	4	1	-5	1	11	c
ligne :	5	1	-5	1	15	d
ligne :	6	1	-5	1	16	e
ligne :	7	1	-5	1	17	f
ligne :	8	1	-5	1	21	g
ligne :	9	1	-5	1	22	h
ligne :	10	1	-5	1	23	i

3) The constant table.

line number		length	value
ligne :	0	4	33.2
ligne :	1	4	447.
ligne :	2	2	34
ligne :	3	1	8
ligne :	4	15	hello the world

4) The parse tree.

```
t_alg 62 INCONNU 0
! t_alh 2 t_alg 62
! ! t_ident 1 t_alh 2
! ! ! alg1 0 t_ident 1
! t_alb 61 t_alg 62
! ! t_exdp 9 t_alb 61
! ! ! t_schdec 3 t_exdp 9
! ! ! t_algdec 4 t_exdp 9
! ! ! t_funcdec 8 t_exdp 9
! ! ! ! t_felt 7 t_funcdec 8
! ! ! ! ! t_ident 5 t_felt 7
! ! ! ! ! ! sin 1 t_ident 5
! ! ! ! ! t_real 6 t_felt 7
! ! t_vadp 28 t_alb 61
! ! ! t_vardec 15 t_vadp 28
! ! ! ! t_names 13 t_vardec 15
! ! ! ! ! t_ident 10 t_names 13
! ! ! ! ! ! a 2 t_ident 10
! ! ! ! ! t_ident 11 t_names 13
! ! ! ! ! ! b 3 t_ident 11
! ! ! ! ! t_ident 12 t_names 13
! ! ! ! ! ! c 4 t_ident 12
! ! ! ! t_real 14 t_vardec 15
! ! ! t_vardec 21 t_vadp 28
! ! ! ! t_names 19 t_vardec 21
! ! ! ! ! t_ident 16 t_names 19
! ! ! ! ! ! d 5 t_ident 16
! ! ! ! ! t_ident 17 t_names 19
! ! ! ! ! ! e 6 t_ident 17
! ! ! ! ! t_ident 18 t_names 19
! ! ! ! ! ! f 7 t_ident 18
! ! ! ! t_int 20 t_vardec 21
! ! ! t_vardec 27 t_vadp 28
! ! ! ! t_names 25 t_vardec 27
! ! ! ! ! t_ident 22 t_names 25
! ! ! ! ! ! g 8 t_ident 22
! ! ! ! ! t_ident 23 t_names 25
! ! ! ! ! ! h 9 t_ident 23
! ! ! ! ! t_ident 24 t_names 25
! ! ! ! ! ! i 10 t_ident 24
! ! ! ! t_str 26 t_vardec 27
```

```
! ! t_stp 60 t_alb 61
! ! ! t_assst 43 t_stp 60
! ! ! ! t_var 30 t_assst 43
! ! ! ! ! t_ident 29 t_var 30
! ! ! ! ! ! a 2 t_ident 29
! ! ! ! t_genexp 42 t_assst 43
! ! ! ! ! t_arexp 41 t_genexp 42
! ! ! ! ! ! t_adexpb 40 t_arexp 41
! ! ! ! ! ! ! t_minus 32 t_adexpb 40
! ! ! ! ! ! ! t_unsnum 31 t_adexpb 40
! ! ! ! ! ! ! ! 33.2 0 t_unsnum 31
! ! ! ! ! ! ! t_arexp 39 t_adexpb 40
! ! ! ! ! ! ! ! t_func 38 t_arexp 39
! ! ! ! ! ! ! ! ! t_ident 33 t_func 38
! ! ! ! ! ! ! ! ! ! sin 1 t_ident 33
! ! ! ! ! ! ! ! ! t_parlis 37 t_func 38
! ! ! ! ! ! ! ! ! ! t_genexp 36 t_parlis 37
! ! ! ! ! ! ! ! ! ! ! t_arexp 35 t_genexp 36
! ! ! ! ! ! ! ! ! ! ! ! t_unsnum 34 t_arexp 35
! ! ! ! ! ! ! ! ! ! ! ! ! 447. 1 t_unsnum 34
! ! ! t_assst 53 t_stp 60
! ! ! ! t_var 45 t_assst 53
! ! ! ! ! t_ident 44 t_var 45
! ! ! ! ! ! d 5 t_ident 44
! ! ! ! t_genexp 52 t_assst 53
! ! ! ! ! t_arexp 51 t_genexp 52
! ! ! ! ! ! t_adexpb 50 t_arexp 51
! ! ! ! ! ! ! t_plus 47 t_adexpb 50
! ! ! ! ! ! ! t_unsnum 46 t_adexpb 50
! ! ! ! ! ! ! ! 34 2 t_unsnum 46
! ! ! ! ! ! ! t_arexp 49 t_adexpb 50
! ! ! ! ! ! ! ! t_unsnum 48 t_arexp 49
! ! ! ! ! ! ! ! ! 8 3 t_unsnum 48
! ! ! t_assst 59 t_stp 60
! ! ! ! t_var 55 t_assst 59
! ! ! ! ! t_ident 54 t_var 55
! ! ! ! ! ! h 9 t_ident 54
! ! ! ! t_genexp 58 t_assst 59
! ! ! ! ! t_arexp 57 t_genexp 58
! ! ! ! ! ! t_str 56 t_arexp 57
! ! ! ! ! ! ! hello the world 4 t_str 56
```

```
algorithm  alg2
algs    print
funcs sin:real
var a,b,c : real
    d,e,f : integer
    g,h,i : string
begin
        a:= - 447.;
        for e := [3:10] do
                if 6 > e then print(e)
                else          print(a)
                endif
        endfor;
end
```

1) The node table.

line number		ident	type	f_son	brother	father
ligne :	0	1	-18	0	-1	1
ligne :	1	2	-6	0	79	80
ligne :	2	3	-88	-1	5	10
ligne :	3	4	-18	1	-1	4
ligne :	4	5	-27	3	-1	5
ligne :	5	6	-11	4	9	10
ligne :	6	7	-18	2	7	8
ligne :	7	8	-21	-1	-1	8
ligne :	8	9	-13	6	-1	9
ligne :	9	10	-12	8	-1	10
ligne :	10	11	-7	2	29	79
ligne :	11	12	-18	3	12	14
ligne :	12	13	-18	4	13	14
ligne :	13	14	-18	5	-1	14
ligne :	14	15	-27	11	15	16
ligne :	15	16	-21	-1	-1	16
ligne :	16	17	-17	14	22	29
ligne :	17	18	-18	6	18	20
ligne :	18	19	-18	7	19	20
ligne :	19	20	-18	8	-1	20
ligne :	20	21	-27	17	21	22
ligne :	21	22	-22	-1	-1	22
ligne :	22	23	-17	20	28	29
ligne :	23	24	-18	9	24	26
ligne :	24	25	-18	10	25	26
ligne :	25	26	-18	11	-1	26

line number		ident	type	f_son	brother	father
ligne :	26	27	-27	23	27	28
ligne :	27	28	-20	-1	-1	28
ligne :	28	29	-17	26	-1	29
ligne :	29	30	-16	16	78	79
ligne :	30	31	-18	3	-1	31
ligne :	31	32	-51	30	36	37
ligne :	32	33	-45	-1	33	34
ligne :	33	34	-54	0	-1	34
ligne :	34	35	-93	32	-1	35
ligne :	35	36	-40	34	-1	36
ligne :	36	37	-35	35	-1	37
ligne :	37	38	-83	31	77	78
ligne :	38	39	-18	7	-1	39
ligne :	39	40	-51	38	50	77
ligne :	40	41	-54	1	-1	41
ligne :	41	42	-40	40	-1	42
ligne :	42	43	-35	41	45	46
ligne :	43	44	-54	2	-1	44
ligne :	44	45	-40	43	-1	45
ligne :	45	46	-35	44	-1	46
ligne :	46	47	-101	42	-1	47
ligne :	47	48	-94	46	-1	48
ligne :	48	49	-40	47	-1	49
ligne :	49	50	-35	48	-1	50
ligne :	50	51	-67	49	76	77
ligne :	51	52	-54	3	-1	52
ligne :	52	53	-40	51	56	57
ligne :	53	54	-58	4	52	57
ligne :	54	55	-18	7	-1	55
ligne :	55	56	-51	54	-1	56
ligne :	56	57	-40	55	-1	57
ligne :	57	58	-39	53	-1	58
ligne :	58	59	-35	57	66	75
ligne :	59	60	-18	1	64	65
ligne :	60	61	-18	7	-1	61

line number		ident	type	f_son	brother	father
ligne :	61	62	-51	60	-1	62
ligne :	62	63	-40	61	-1	63
ligne :	63	64	-35	62	-1	64
ligne :	64	65	-87	63	-1	65
ligne :	65	66	-82	59	-1	66
ligne :	66	67	-90	65	74	75
ligne :	67	68	-18	1	72	73
ligne :	68	69	-18	3	-1	69
ligne :	69	70	-51	68	-1	70
ligne :	70	71	-40	69	-1	71
ligne :	71	72	-35	70	-1	72
ligne :	72	73	-87	71	-1	73
ligne :	73	74	-82	67	-1	74
ligne :	74	75	-90	73	-1	75
ligne :	75	76	-80	58	-1	76
ligne :	76	77	-90	75	-1	77
ligne :	77	78	-66	39	-1	78
ligne :	78	79	-65	37	-1	79
ligne :	79	80	-25	10	-1	80
ligne :	80	81	-5	1	-1	-1

2) The symbol table.

line number		length	class	level	tree	value
ligne :	0	4	-2	1	0	alg2
ligne :	1	5	-2	1	3	print
ligne :	2	3	-4	1	6	sin
ligne :	3	1	-5	1	11	a
ligne :	4	1	-5	1	12	b
ligne :	5	1	-5	1	13	c
ligne :	6	1	-5	1	17	d
ligne :	7	1	-5	1	18	e
ligne :	8	1	-5	1	19	f
ligne :	9	1	-5	1	23	g
ligne :	10	1	-5	1	24	h
ligne :	11	1	-5	1	25	i

3) The constant table.

line number		length	value
ligne :	0	4	447.
ligne :	1	1	3
ligne :	2	2	10
ligne :	3	1	6
ligne :	4	1	>

4) The parse tree.

```
t_alg 81 INCONNU 0
! t_alh 2 t_alg 81
! ! t_ident 1 t_alh 2
! ! ! alg2 0 t_ident 1
! t_alb 80 t_alg 81
! ! t_exdp 11 t_alb 80
! ! ! t_schdec 3 t_exdp 11
! ! ! t_algdec 6 t_exdp 11
! ! ! ! t_names 5 t_algdec 6
! ! ! ! ! t_ident 4 t_names 5
! ! ! ! ! ! print 1 t_ident 4
! ! ! t_funcdec 10 t_exdp 11
! ! ! ! t_felt 9 t_funcdec 10
! ! ! ! ! t_ident 7 t_felt 9
! ! ! ! ! ! sin 2 t_ident 7
! ! ! ! ! t_real 8 t_felt 9
! ! t_vadp 30 t_alb 80
! ! ! t_vardec 17 t_vadp 30
! ! ! ! t_names 15 t_vardec 17
! ! ! ! ! t_ident 12 t_names 15
! ! ! ! ! ! a 3 t_ident 12
! ! ! ! ! t_ident 13 t_names 15
! ! ! ! ! ! b 4 t_ident 13
! ! ! ! ! t_ident 14 t_names 15
! ! ! ! ! ! c 5 t_ident 14
! ! ! ! t_real 16 t_vardec 17
! ! ! t_vardec 23 t_vadp 30
! ! ! ! t_names 21 t_vardec 23
! ! ! ! ! t_ident 18 t_names 21
! ! ! ! ! ! d 6 t_ident 18
! ! ! ! ! t_ident 19 t_names 21
! ! ! ! ! ! e 7 t_ident 19
! ! ! ! ! t_ident 20 t_names 21
! ! ! ! ! ! f 8 t_ident 20
! ! ! ! t_int 22 t_vardec 23
! ! ! t_vardec 29 t_vadp 30
! ! ! ! t_names 27 t_vardec 29
! ! ! ! ! t_ident 24 t_names 27
! ! ! ! ! ! g 9 t_ident 24
! ! ! ! ! t_ident 25 t_names 27
! ! ! ! ! ! h 10 t_ident 25
! ! ! ! ! t_ident 26 t_names 27
! ! ! ! ! ! i 11 t_ident 26
! ! ! ! t_str 28 t_vardec 29
```

```
! ! t_stp 79 t_alb 80
! ! ! t_assst 38 t_stp 79
! ! ! ! t_var 32 t_assst 38
! ! ! ! ! t_ident 31 t_var 32
! ! ! ! ! ! a 3 t_ident 31
! ! ! ! t_genexp 37 t_assst 38
! ! ! ! ! t_arexp 36 t_genexp 37
! ! ! ! ! ! t_adexpu 35 t_arexp 36
! ! ! ! ! ! ! t_minus 33 t_adexpu 35
! ! ! ! ! ! ! t_unsnum 34 t_adexpu 35
! ! ! ! ! ! ! ! 447. 0 t_unsnum 34
! ! ! t_enumloop 78 t_stp 79
! ! ! ! t_var 40 t_enumloop 78
! ! ! ! ! t_ident 39 t_var 40
! ! ! ! ! ! e 7 t_ident 39
! ! ! ! t_forlis 51 t_enumloop 78
! ! ! ! ! t_genexp 50 t_forlis 51
! ! ! ! ! ! t_arexp 49 t_genexp 50
! ! ! ! ! ! ! t_collexp 48 t_arexp 49
! ! ! ! ! ! ! ! t_rangeb 47 t_collexp 48
! ! ! ! ! ! ! ! ! t_genexp 43 t_rangeb 47
! ! ! ! ! ! ! ! ! ! t_arexp 42 t_genexp 43
! ! ! ! ! ! ! ! ! ! ! t_unsnum 41 t_arexp 42
! ! ! ! ! ! ! ! ! ! ! ! 3 1 t_unsnum 41
! ! ! ! ! ! ! ! ! t_genexp 46 t_rangeb 47
! ! ! ! ! ! ! ! ! ! t_arexp 45 t_genexp 46
! ! ! ! ! ! ! ! ! ! ! t_unsnum 44 t_arexp 45
! ! ! ! ! ! ! ! ! ! ! ! 10 2 t_unsnum 44
! ! ! ! t_actions 77 t_enumloop 78
! ! ! ! ! t_altern 76 t_actions 77
! ! ! ! ! ! t_genexp 59 t_altern 76
! ! ! ! ! ! ! t_tesexp 58 t_genexp 59
! ! ! ! ! ! ! ! t_tesop 54 t_tesexp 58
! ! ! ! ! ! ! ! ! > 4 t_tesop 54
! ! ! ! ! ! ! ! t_arexp 53 t_tesexp 58
! ! ! ! ! ! ! ! ! t_unsnum 52 t_arexp 53
! ! ! ! ! ! ! ! ! ! 6 3 t_unsnum 52
! ! ! ! ! ! ! ! t_arexp 57 t_tesexp 58
! ! ! ! ! ! ! ! ! t_var 56 t_arexp 57
! ! ! ! ! ! ! ! ! ! t_ident 55 t_var 56
! ! ! ! ! ! ! ! ! ! ! e 7 t_ident 55
! ! ! ! ! ! t_actions 67 t_altern 76
! ! ! ! ! ! ! t_call 66 t_actions 67
! ! ! ! ! ! ! ! t_ident 60 t_call 66
! ! ! ! ! ! ! ! ! print 1 t_ident 60
! ! ! ! ! ! ! ! t_parlis 65 t_call 66
! ! ! ! ! ! ! ! ! t_genexp 64 t_parlis 65
! ! ! ! ! ! ! ! ! ! t_arexp 63 t_genexp 64
! ! ! ! ! ! ! ! ! ! ! t_var 62 t_arexp 63
! ! ! ! ! ! ! ! ! ! ! ! t_ident 61 t_var 62
! ! ! ! ! ! ! ! ! ! ! ! ! e 7 t_ident 61
! ! ! ! ! ! t_actions 75 t_altern 76
! ! ! ! ! ! ! t_call 74 t_actions 75
! ! ! ! ! ! ! ! t_ident 68 t_call 74
! ! ! ! ! ! ! ! ! print 1 t_ident 68
! ! ! ! ! ! ! ! t_parlis 73 t_call 74
! ! ! ! ! ! ! ! ! t_genexp 72 t_parlis 73
! ! ! ! ! ! ! ! ! ! t_arexp 71 t_genexp 72
! ! ! ! ! ! ! ! ! ! ! t_var 70 t_arexp 71
! ! ! ! ! ! ! ! ! ! ! ! t_ident 69 t_var 70
! ! ! ! ! ! ! ! ! ! ! ! ! a 3 t_ident 69
```

```
algorithm efectiv1

schema  example1

funcs addlist: list of record CUSTOMER,
      print : integer

var LINE_OK :boolean
    CUS : record CUSTOMER
    LINE : record ORDER_LINE
    PRO : record PRODUCT
    O : record ORDER
    LIST_CUSTOMER : list of record CUSTOMER

begin
for  LINE := ORDER_LINE  do
     LINE_OK := false;

     for PRO := PRODUCT(PROLI : LINE) do
         if NUM(:PRO) = 783 then
                LINE_OK := true;
                exit PRO
         endif
     endfor
endfor;

if LINE_OK then

     for  O := ORDER(ORLI : LINE) do
        for CUS := CUSTOMER(CUSOR : O) do
            if CUS in LIST_CUSTOMER then
                        addlist(CUS,LIST_CUSTOMER);
                        print(NAME(:CUS))
            endif
        endfor
     endfor
endif
end
```

1) The node table.

line number		ident	type	f_son	brother	father
ligne :	0	1	-18	0	-1	1
ligne :	1	2	-6	0	180	181
ligne :	2	3	-18	1	-1	3
ligne :	3	4	-88	2	4	14
ligne :	4	5	-11	-1	13	14
ligne :	5	6	-18	15	8	12
ligne :	6	7	-18	2	-1	7
ligne :	7	8	-23	6	-1	8
ligne :	8	9	-89	7	-1	12
ligne :	9	10	-18	16	10	11
ligne :	10	11	-22	-1	-1	11
ligne :	11	12	-13	9	-1	13
ligne :	12	13	-13	5	11	13
ligne :	13	14	-12	12	-1	14
ligne :	14	15	-7	3	45	180
ligne :	15	16	-18	17	-1	16
ligne :	16	17	-27	15	17	18
ligne :	17	18	-19	-1	-1	18
ligne :	18	19	-17	16	23	45
ligne :	19	20	-18	18	-1	20
ligne :	20	21	-27	19	22	23
ligne :	21	22	-18	2	-1	22
ligne :	22	23	-23	21	-1	23
ligne :	23	24	-17	20	28	45
ligne :	24	25	-18	19	-1	25
ligne :	25	26	-27	24	27	28
ligne :	26	27	-18	4	-1	27
ligne :	27	28	-23	26	-1	28
ligne :	28	29	-17	25	33	45
ligne :	29	30	-18	20	-1	30
ligne :	30	31	-27	29	32	33
ligne :	31	32	-18	5	-1	32
ligne :	32	33	-23	31	-1	33
ligne :	33	34	-17	30	38	45
ligne :	34	35	-18	21	-1	35
ligne :	35	36	-27	34	37	38
ligne :	36	37	-18	3	-1	37
ligne :	37	38	-23	36	-1	38
ligne :	38	39	-17	35	44	45
ligne :	39	40	-18	22	-1	40
ligne :	40	41	-27	39	43	44
ligne :	41	42	-18	2	-1	42
ligne :	42	43	-23	41	-1	43
ligne :	43	44	-89	42	-1	44
ligne :	44	45	-17	40	-1	45
ligne :	45	46	-16	18	179	180
ligne :	46	47	-18	19	-1	47
ligne :	47	48	-51	46	52	100
ligne :	48	49	-18	4	-1	49
ligne :	49	50	-51	48	-1	50

line number		ident	type	f_son	brother	father
ligne :	50	51	-40	49	-1	51
ligne :	51	52	-35	50	-1	52
ligne :	52	53	-67	51	99	100
ligne :	53	54	-18	17	-1	54
ligne :	54	55	-51	53	56	57
ligne :	55	56	-56	0	-1	56
ligne :	56	57	-35	55	-1	57
ligne :	57	58	-83	54	98	99
ligne :	58	59	-18	20	-1	59
ligne :	59	60	-51	58	72	98
ligne :	60	61	-18	5	67	68
ligne :	61	62	-18	14	-1	63
ligne :	62	63	-57	-1	61	63
ligne :	63	64	-49	62	65	66
ligne :	64	65	-18	19	-1	65
ligne :	65	66	-51	64	-1	66
ligne :	66	67	-63	63	-1	67
ligne :	67	68	-97	66	-1	68
ligne :	68	69	-96	60	-1	69
ligne :	69	70	-94	68	-1	70
ligne :	70	71	-40	69	-1	71
ligne :	71	72	-35	70	-1	72
ligne :	72	73	-67	71	97	98
ligne :	73	74	-18	7	79	80
ligne :	74	75	-57	-1	-1	75
ligne :	75	76	-49	74	77	78
ligne :	76	77	-18	20	-1	77
ligne :	77	78	-51	76	-1	78
ligne :	78	79	-63	75	-1	79
ligne :	79	80	-97	78	-1	80
ligne :	80	81	-96	73	-1	81
ligne :	81	82	-94	80	-1	82
ligne :	82	83	-40	81	85	86
ligne :	83	84	-58	1	82	86
ligne :	84	85	-54	2	-1	85
ligne :	85	86	-40	84	-1	86
ligne :	86	87	-39	83	-1	87
ligne :	87	88	-35	86	95	96
ligne :	88	89	-18	17	-1	89
ligne :	89	90	-51	88	91	92
ligne :	90	91	-56	3	-1	91
ligne :	91	92	-35	90	-1	92
ligne :	92	93	-83	89	94	95
ligne :	93	94	-18	20	-1	94
ligne :	94	95	-91	93	-1	95
ligne :	95	96	-90	92	-1	96
ligne :	96	97	-80	87	-1	97
ligne :	97	98	-90	96	-1	98
ligne :	98	99	-66	59	-1	99
ligne :	99	100	-90	57	-1	100

line number		ident	type	f_son	brother	father
ligne :	100	101	-66	47	178	179
ligne :	101	102	-18	17	-1	102
ligne :	102	103	-51	101	-1	103
ligne :	103	104	-40	102	-1	104
ligne :	104	105	-35	103	177	178
ligne :	105	106	-18	21	-1	106
ligne :	106	107	-51	105	119	176
ligne :	107	108	-18	3	114	115
ligne :	108	109	-18	11	-1	110
ligne :	109	110	-57	-1	108	110
ligne :	110	111	-49	109	112	113
ligne :	111	112	-18	19	-1	112
ligne :	112	113	-51	111	-1	113
ligne :	113	114	-63	110	-1	114
ligne :	114	115	-97	113	-1	115
ligne :	115	116	-96	107	-1	116
ligne :	116	117	-94	115	-1	117
ligne :	117	118	-40	116	-1	118
ligne :	118	119	-35	117	-1	119
ligne :	119	120	-67	118	175	176
ligne :	120	121	-18	18	-1	121
ligne :	121	122	-51	120	134	174
ligne :	122	123	-18	2	129	130
ligne :	123	124	-18	9	-1	125
ligne :	124	125	-57	-1	123	125
ligne :	125	126	-49	124	127	128
ligne :	126	127	-18	21	-1	127
ligne :	127	128	-51	126	-1	128
ligne :	128	129	-63	125	-1	129
ligne :	129	130	-97	128	-1	130
ligne :	130	131	-96	122	-1	131
ligne :	131	132	-94	130	-1	132
ligne :	132	133	-40	131	-1	133
ligne :	133	134	-35	132	-1	134
ligne :	134	135	-67	133	173	174
ligne :	135	136	-18	18	141	142
ligne :	136	137	-58	4	139	140
ligne :	137	138	-18	22	-1	138
ligne :	138	139	-51	137	-1	139
ligne :	139	140	-40	138	-1	-1
ligne :	140	141	-50	136	-1	141
ligne :	141	142	-97	140	-1	142
ligne :	142	143	-96	135	-1	143
ligne :	143	144	-94	142	-1	144
ligne :	144	145	-40	143	-1	145
ligne :	145	146	-35	144	171	172
ligne :	146	147	-18	15	155	156
ligne :	147	148	-18	18	-1	148
ligne :	148	149	-51	147	-1	149
ligne :	149	150	-40	148	-1	150

line number		ident	type	f_son	brother	father
ligne :	150	151	-35	149	154	155
ligne :	151	152	-18	22	-1	152
ligne :	152	153	-51	151	-1	153
ligne :	153	154	-40	152	-1	154
ligne :	154	155	-35	153	-1	155
ligne :	155	156	-87	150	-1	156
ligne :	156	157	-82	146	170	171
ligne :	157	158	-18	16	169	170
ligne :	158	159	-18	6	164	165
ligne :	159	160	-57	-1	-1	160
ligne :	160	161	-49	159	162	163
ligne :	161	162	-18	18	-1	162
ligne :	162	163	-51	161	-1	163
ligne :	163	164	-63	160	-1	164
ligne :	164	165	-97	163	-1	165
ligne :	165	166	-96	158	-1	166
ligne :	166	167	-94	165	-1	167
ligne :	167	168	-40	166	-1	168
ligne :	168	169	-35	167	-1	169
ligne :	169	170	-87	168	-1	170
ligne :	170	171	-82	157	-1	171
ligne :	171	172	-90	156	-1	172
ligne :	172	173	-80	145	-1	173
ligne :	173	174	-90	172	-1	174
ligne :	174	175	-66	121	-1	175
ligne :	175	176	-90	174	-1	176
ligne :	176	177	-66	106	-1	177
ligne :	177	178	-90	176	-1	178
ligne :	178	179	-80	104	-1	179
ligne :	179	180	-65	100	-1	180
ligne :	180	181	-25	14	-1	181
ligne :	181	182	-5	1	-1	-1

2) The symbol table.

line number		length	class	level	tree	value
ligne :	0	8	-2	1	0	efectiv1
ligne :	1	8	-1	1	2	example1
ligne :	2	8	-3	1	-1	CUSTOMER
ligne :	3	5	-3	1	-1	ORDER
ligne :	4	10	-3	1	-1	ORDER_LINE
ligne :	5	7	-3	1	-1	PRODUCT
ligne :	6	4	-6	1	-1	NAME
ligne :	7	3	-6	1	-1	NUM
ligne :	8	8	-6	1	-1	QUANTITY
ligne :	9	5	-7	1	-1	CUSOR
ligne :	10	5	-7	1	-1	ORCUS
ligne :	11	4	-7	1	-1	ORLI
ligne :	12	4	-7	1	-1	LIOR
ligne :	13	5	-7	1	-1	LIPRO
ligne :	14	5	-7	1	-1	PROLI
ligne :	15	7	-4	1	5	addlist
ligne :	16	5	-4	1	9	print
ligne :	17	7	-5	1	15	LINE_OK
ligne :	18	3	-5	1	19	CUS
ligne :	19	4	-5	1	24	LINE
ligne :	20	3	-5	1	29	PRO
ligne :	21	1	-5	1	34	O
ligne :	22	13	-5	1	39	LIST_CUSTOMER

3) The constant table.

line number		length	value
ligne :	0	5	false
ligne :	1	1	=
ligne :	2	3	783
ligne :	3	4	true
ligne :	4	2	in

4) The parse tree

```
t_alg 182 INCONNU 0
! t_alh 2 t_alg 182
! ! t_ident 1 t_alh 2
! ! ! efectiv1 0 t_ident 1
! t_alb 181 t_alg 182
! ! t_exdp 15 t_alb 181
! ! ! t_schdec 4 t_exdp 15
! ! ! ! t_ident 3 t_schdec 4
! ! ! ! ! example1 1 t_ident 3
! ! ! t_algdec 5 t_exdp 15
! ! ! t_funcdec 14 t_exdp 15
! ! ! ! t_felt 13 t_funcdec 14
! ! ! ! ! t_ident 6 t_felt 13
! ! ! ! ! ! addlist 15 t_ident 6
! ! ! ! ! t_listyp 9 t_felt 13
! ! ! ! ! ! t_rectyp 8 t_listyp 9
! ! ! ! ! ! ! t_ident 7 t_rectyp 8
! ! ! ! ! ! ! ! CUSTOMER 2 t_ident 7
! ! ! ! t_felt 12 t_funcdec 14
! ! ! ! ! t_ident 10 t_felt 12
! ! ! ! ! ! print 16 t_ident 10
! ! ! ! ! t_int 11 t_felt 12
! ! t_vadp 46 t_alb 181
! ! ! t_vardec 19 t_vadp 46
! ! ! ! t_names 17 t_vardec 19
! ! ! ! ! t_ident 16 t_names 17
! ! ! ! ! ! LINE_OK 17 t_ident 16
! ! ! ! t_bool 18 t_vardec 19
! ! ! t_vardec 24 t_vadp 46
! ! ! ! t_names 21 t_vardec 24
! ! ! ! ! t_ident 20 t_names 21
! ! ! ! ! ! CUS 18 t_ident 20
! ! ! ! t_rectyp 23 t_vardec 24
! ! ! ! ! t_ident 22 t_rectyp 23
! ! ! ! ! ! CUSTOMER 2 t_ident 22
! ! ! t_vardec 29 t_vadp 46
! ! ! ! t_names 26 t_vardec 29
! ! ! ! ! t_ident 25 t_names 26
! ! ! ! ! ! LINE 19 t_ident 25
! ! ! ! t_rectyp 28 t_vardec 29
! ! ! ! ! t_ident 27 t_rectyp 28
! ! ! ! ! ! ORDER_LINE 4 t_ident 27
! ! ! t_vardec 34 t_vadp 46
! ! ! ! t_names 31 t_vardec 34
! ! ! ! ! t_ident 30 t_names 31
! ! ! ! ! ! PRO 20 t_ident 30
! ! ! ! t_rectyp 33 t_vardec 34
! ! ! ! ! t_ident 32 t_rectyp 33
! ! ! ! ! ! PRODUCT 5 t_ident 32
! ! ! t_vardec 39 t_vadp 46
! ! ! ! t_names 36 t_vardec 39
! ! ! ! ! t_ident 35 t_names 36
! ! ! ! ! ! O 21 t_ident 35
! ! ! ! t_rectyp 38 t_vardec 39
! ! ! ! ! t_ident 37 t_rectyp 38
! ! ! ! ! ! ORDER 3 t_ident 37
! ! ! t_vardec 45 t_vadp 46
! ! ! ! t_names 41 t_vardec 45
! ! ! ! ! t_ident 40 t_names 41
! ! ! ! ! ! LIST_CUSTOMER 22 t_ident 40
! ! ! ! t_listyp 44 t_vardec 45
! ! ! ! ! t_rectyp 43 t_listyp 44
! ! ! ! ! ! t_ident 42 t_rectyp 43
! ! ! ! ! ! ! CUSTOMER 2 t_ident 42
```

```
! ! t_stp 180 t_alb 181
! ! ! t_enumloop 101 t_stp 180
! ! ! ! t_var 48 t_enumloop 101
! ! ! ! ! t_ident 47 t_var 48
! ! ! ! ! ! LINE 19 t_ident 47
! ! ! ! t_forlis 53 t_enumloop 101
! ! ! ! ! t_genexp 52 t_forlis 53
! ! ! ! ! ! t_arexp 51 t_genexp 52
! ! ! ! ! ! ! t_var 50 t_arexp 51
! ! ! ! ! ! ! ! t_ident 49 t_var 50
! ! ! ! ! ! ! ! ! ORDER_LINE 4 t_ident 49
! ! ! ! t_actions 100 t_enumloop 101
! ! ! ! ! t_assst 58 t_actions 100
! ! ! ! ! ! t_var 55 t_assst 58
! ! ! ! ! ! ! t_ident 54 t_var 55
! ! ! ! ! ! ! ! LINE_OK 17 t_ident 54
! ! ! ! ! ! t_genexp 57 t_assst 58
! ! ! ! ! ! ! t_logval 56 t_genexp 57
! ! ! ! ! ! ! ! false 0 t_logval 56
! ! ! ! ! t_enumloop 99 t_actions 100
! ! ! ! ! ! t_var 60 t_enumloop 99
! ! ! ! ! ! ! t_ident 59 t_var 60
! ! ! ! ! ! ! ! PRO 20 t_ident 59
! ! ! ! ! ! t_forlis 73 t_enumloop 99
! ! ! ! ! ! ! t_genexp 72 t_forlis 73
! ! ! ! ! ! ! ! t_arexp 71 t_genexp 72
! ! ! ! ! ! ! ! ! t_collexp 70 t_arexp 71
! ! ! ! ! ! ! ! ! ! t_dbset 69 t_collexp 70
! ! ! ! ! ! ! ! ! ! ! t_ident 61 t_dbset 69
! ! ! ! ! ! ! ! ! ! ! ! PRODUCT 5 t_ident 61
! ! ! ! ! ! ! ! ! ! ! t_predicate 68 t_dbset 69
! ! ! ! ! ! ! ! ! ! ! ! t_relcon 67 t_predicate 68
! ! ! ! ! ! ! ! ! ! ! ! ! t_relop 64 t_relcon 67
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_co 63 t_relop 64
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 62 t_relop 64
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! PROLI 14 t_ident 62
! ! ! ! ! ! ! ! ! ! ! ! ! t_var 66 t_relcon 67
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 65 t_var 66
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! LINE 19 t_ident 65
! ! ! ! ! ! t_actions 98 t_enumloop 99
! ! ! ! ! ! ! t_altern 97 t_actions 98
! ! ! ! ! ! ! ! t_genexp 88 t_altern 97
! ! ! ! ! ! ! ! ! t_tesexp 87 t_genexp 88
! ! ! ! ! ! ! ! ! ! t_tesop 84 t_tesexp 87
! ! ! ! ! ! ! ! ! ! ! = 1 t_tesop 84
! ! ! ! ! ! ! ! ! ! t_arexp 83 t_tesexp 87
! ! ! ! ! ! ! ! ! ! ! t_collexp 82 t_arexp 83
! ! ! ! ! ! ! ! ! ! ! ! t_dbset 81 t_collexp 82
! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 74 t_dbset 81
! ! ! ! ! ! ! ! ! ! ! ! ! ! NUM 7 t_ident 74
! ! ! ! ! ! ! ! ! ! ! ! ! t_predicate 80 t_dbset 81
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_relcon 79 t_predicate 80
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_relop 76 t_relcon 79
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_co 75 t_relop 76
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_var 78 t_relcon 79
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 77 t_var 78
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! PRO 20 t_ident 77
```

```
! ! ! ! ! ! ! ! ! ! t_arexp 86 t_tesexp 87
! ! ! ! ! ! ! ! ! ! ! t_unsnum 85 t_arexp 86
! ! ! ! ! ! ! ! ! ! ! ! 783 2 t_unsnum 85
! ! ! ! ! ! ! ! t_actions 96 t_altern 97
! ! ! ! ! ! ! ! ! t_assst 93 t_actions 96
! ! ! ! ! ! ! ! ! ! t_var 90 t_assst 93
! ! ! ! ! ! ! ! ! ! ! t_ident 89 t_var 90
! ! ! ! ! ! ! ! ! ! ! ! LINE_OK 17 t_ident 89
! ! ! ! ! ! ! ! ! ! t_genexp 92 t_assst 93
! ! ! ! ! ! ! ! ! ! ! t_logval 91 t_genexp 92
! ! ! ! ! ! ! ! ! ! ! ! true 3 t_logval 91
! ! ! ! ! ! ! ! ! t_exit 95 t_actions 96
! ! ! ! ! ! ! ! ! ! t_ident 94 t_exit 95
! ! ! ! ! ! ! ! ! ! ! PRO 20 t_ident 94
! ! ! t_altern 179 t_stp 180
! ! ! ! t_genexp 105 t_altern 179
! ! ! ! ! t_arexp 104 t_genexp 105
! ! ! ! ! ! t_var 103 t_arexp 104
! ! ! ! ! ! ! t_ident 102 t_var 103
! ! ! ! ! ! ! ! LINE_OK 17 t_ident 102
! ! ! ! t_actions 178 t_altern 179
! ! ! ! ! t_enumloop 177 t_actions 178
! ! ! ! ! ! t_var 107 t_enumloop 177
! ! ! ! ! ! ! t_ident 106 t_var 107
! ! ! ! ! ! ! ! O 21 t_ident 106
! ! ! ! ! ! t_forlis 120 t_enumloop 177
! ! ! ! ! ! ! t_genexp 119 t_forlis 120
! ! ! ! ! ! ! ! t_arexp 118 t_genexp 119
! ! ! ! ! ! ! ! ! t_collexp 117 t_arexp 118
! ! ! ! ! ! ! ! ! ! t_dbset 116 t_collexp 117
! ! ! ! ! ! ! ! ! ! ! t_ident 108 t_dbset 116
! ! ! ! ! ! ! ! ! ! ! ! ORDER 3 t_ident 108
! ! ! ! ! ! ! ! ! ! ! t_predicate 115 t_dbset 116
! ! ! ! ! ! ! ! ! ! ! ! t_relcon 114 t_predicate 115
! ! ! ! ! ! ! ! ! ! ! ! ! t_relop 111 t_relcon 114
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_co 110 t_relop 111
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 109 t_relop 111
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ORLI 11 t_ident 109
! ! ! ! ! ! ! ! ! ! ! ! ! t_var 113 t_relcon 114
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 112 t_var 113
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! LINE 19 t_ident 112
! ! ! ! ! ! t_actions 176 t_enumloop 177
! ! ! ! ! ! ! t_enumloop 175 t_actions 176
! ! ! ! ! ! ! ! t_var 122 t_enumloop 175
! ! ! ! ! ! ! ! ! t_ident 121 t_var 122
! ! ! ! ! ! ! ! ! ! CUS 18 t_ident 121
! ! ! ! ! ! ! ! t_forlis 135 t_enumloop 175
! ! ! ! ! ! ! ! ! t_genexp 134 t_forlis 135
! ! ! ! ! ! ! ! ! ! t_arexp 133 t_genexp 134
! ! ! ! ! ! ! ! ! ! ! t_collexp 132 t_arexp 133
! ! ! ! ! ! ! ! ! ! ! ! t_dbset 131 t_collexp 132
! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 123 t_dbset 131
! ! ! ! ! ! ! ! ! ! ! ! ! ! CUSTOMER 2 t_ident 123
! ! ! ! ! ! ! ! ! ! ! ! ! t_predicate 130 t_dbset 131
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_relcon 129 t_predicate 130
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_relop 126 t_relcon 129
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_co 125 t_relop 126
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 124 t_relop 126
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! CUSOR 9 t_ident 124
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_var 128 t_relcon 129
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 127 t_var 128
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! O 21 t_ident 127
```

```
! ! ! ! ! ! ! ! t_actions 174 t_enumloop 175
! ! ! ! ! ! ! ! ! t_altern 173 t_actions 174
! ! ! ! ! ! ! ! ! ! t_genexp 146 t_altern 173
! ! ! ! ! ! ! ! ! ! ! t_arexp 145 t_genexp 146
! ! ! ! ! ! ! ! ! ! ! ! t_collexp 144 t_arexp 145
! ! ! ! ! ! ! ! ! ! ! ! ! t_dbset 143 t_collexp 144
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 136 t_dbset 143
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! CUS 18 t_ident 136
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_predicate 142 t_dbset 143
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_belcon 141 t_predicate 142
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_tesop 137 t_belcon 141
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! in 4 t_tesop 137
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_arexp 140 INCONNU 0
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_var 139 t_arexp 140
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 138 t_var 139
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! LIST_CUSTOMER 22 t_ident 138
! ! ! ! ! ! ! ! ! ! t_actions 172 t_altern 173
! ! ! ! ! ! ! ! ! ! ! t_call 157 t_actions 172
! ! ! ! ! ! ! ! ! ! ! ! t_ident 147 t_call 157
! ! ! ! ! ! ! ! ! ! ! ! ! addlist 15 t_ident 147
! ! ! ! ! ! ! ! ! ! ! ! t_parlis 156 t_call 157
! ! ! ! ! ! ! ! ! ! ! ! ! t_genexp 151 t_parlis 156
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_arexp 150 t_genexp 151
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_var 149 t_arexp 150
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 148 t_var 149
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! CUS 18 t_ident 148
! ! ! ! ! ! ! ! ! ! ! ! ! t_genexp 155 t_parlis 156
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_arexp 154 t_genexp 155
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_var 153 t_arexp 154
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 152 t_var 153
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! LIST_CUSTOMER 22 t_ident 152
! ! ! ! ! ! ! ! ! ! ! t_call 171 t_actions 172
! ! ! ! ! ! ! ! ! ! ! ! t_ident 158 t_call 171
! ! ! ! ! ! ! ! ! ! ! ! ! print 16 t_ident 158
! ! ! ! ! ! ! ! ! ! ! ! t_parlis 170 t_call 171
! ! ! ! ! ! ! ! ! ! ! ! ! t_genexp 169 t_parlis 170
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_arexp 168 t_genexp 169
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_collexp 167 t_arexp 168
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_dbset 166 t_collexp 167
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 159 t_dbset 166
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! NAME 6 t_ident 159
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_predicate 165 t_dbset 166
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_relcon 164 t_predicate 165
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_relop 161 t_relcon 164
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_co 160 t_relop 161
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_var 163 t_relcon 164
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 162 t_var 163
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! CUS 18 t_ident 162
```

```
algorithm  alg1
var a,b,c : real
    d : structure
        test : string
        d1 : structure
                test : integer
                d11 : real
                d12 : structure
                        test : boolean
                        d121 : integer
                        d122 : real
                      end
                d13 : boolean
                d14 : structure
                        test : list of string
                        d141 : list of integer
                      end
             end
        d2: boolean
        end
    g,h,i : string
begin
        d.test := 'hello';
        d.d1.test := 3;
        d.d1.d11 := -3.;
        d.d1.d12.test := true;
        d.d1.d12.d121 := 4;
        d.d1.d12.d122 := 4.;
        d.d1.d14.d141 := {1,2,4,6,8};
        d.d2 := false;
        h:=d.test
end
```

1) The node table.

line number		ident	type	f_son	brother	father
ligne :	0	1	-18	0	-1	1
ligne :	1	2	-6	0	177	178
ligne :	2	3	-88	-1	3	5
ligne :	3	4	-11	-1	4	5
ligne :	4	5	-12	-1	-1	5
ligne :	5	6	-7	2	76	177
ligne :	6	7	-18	1	7	9
ligne :	7	8	-18	2	8	9
ligne :	8	9	-18	3	-1	9
ligne :	9	10	-27	6	10	11
ligne :	10	11	-21	-1	-1	11
ligne :	11	12	-17	9	69	76
ligne :	12	13	-18	4	-1	13
ligne :	13	14	-27	12	68	69
ligne :	14	15	-18	5	-1	15
ligne :	15	16	-27	14	16	17

line number		ident	type	f_son	brother	father
ligne :	16	17	-20	-1	-1	17
ligne :	17	18	-17	15	63	68
ligne :	18	19	-18	6	-1	19
ligne :	19	20	-27	18	62	63
ligne :	20	21	-18	7	-1	21
ligne :	21	22	-27	20	22	23
ligne :	22	23	-22	-1	-1	23
ligne :	23	24	-17	21	27	62
ligne :	24	25	-18	8	-1	25
ligne :	25	26	-27	24	26	27
ligne :	26	27	-21	-1	-1	27
ligne :	27	28	-17	25	43	62
ligne :	28	29	-18	9	-1	29
ligne :	29	30	-27	28	42	43
ligne :	30	31	-18	10	-1	31
ligne :	31	32	-27	30	32	33
ligne :	32	33	-19	-1	-1	33
ligne :	33	34	-17	31	37	42
ligne :	34	35	-18	11	-1	35
ligne :	35	36	-27	34	36	37
ligne :	36	37	-22	-1	-1	37
ligne :	37	38	-17	35	41	42
ligne :	38	39	-18	12	-1	39
ligne :	39	40	-27	38	40	41
ligne :	40	41	-21	-1	-1	41
ligne :	41	42	-17	39	-1	42
ligne :	42	43	-26	33	-1	43
ligne :	43	44	-17	29	47	62
ligne :	44	45	-18	13	-1	45
ligne :	45	46	-27	44	46	47
ligne :	46	47	-19	-1	-1	47
ligne :	47	48	-17	45	61	62
ligne :	48	49	-18	14	-1	49
ligne :	49	50	-27	48	60	61
ligne :	50	51	-18	15	-1	51
ligne :	51	52	-27	50	53	54
ligne :	52	53	-20	-1	-1	53
ligne :	53	54	-89	52	-1	54
ligne :	54	55	-17	51	59	60
ligne :	55	56	-18	16	-1	56
ligne :	56	57	-27	55	58	59
ligne :	57	58	-22	-1	-1	58
ligne :	58	59	-89	57	-1	59
ligne :	59	60	-17	56	-1	60
ligne :	60	61	-26	54	-1	61
ligne :	61	62	-17	49	-1	62
ligne :	62	63	-26	23	-1	63
ligne :	63	64	-17	19	67	68
ligne :	64	65	-18	17	-1	65
ligne :	65	66	-27	64	66	67
ligne :	66	67	-19	-1	-1	67

line number		ident	type	f_son	brother	father
ligne :	67	68	-17	65	-1	68
ligne :	68	69	-26	17	-1	69
ligne :	69	70	-17	13	75	76
ligne :	70	71	-18	18	71	73
ligne :	71	72	-18	19	72	73
ligne :	72	73	-18	20	-1	73
ligne :	73	74	-27	70	74	75
ligne :	74	75	-20	-1	-1	75
ligne :	75	76	-17	73	-1	76
ligne :	76	77	-16	11	176	177
ligne :	77	78	-18	4	78	79
ligne :	78	79	-18	5	-1	79
ligne :	79	80	-53	77	-1	80
ligne :	80	81	-51	79	83	84
ligne :	81	82	-2	0	-1	82
ligne :	82	83	-40	81	-1	83
ligne :	83	84	-35	82	-1	84
ligne :	84	85	-83	80	93	176
ligne :	85	86	-18	4	86	88
ligne :	86	87	-18	6	87	88
ligne :	87	88	-18	7	-1	88
ligne :	88	89	-53	85	-1	89
ligne :	89	90	-51	88	92	93
ligne :	90	91	-54	1	-1	91
ligne :	91	92	-40	90	-1	92
ligne :	92	93	-35	91	-1	93
ligne :	93	94	-83	89	104	176
ligne :	94	95	-18	4	95	97
ligne :	95	96	-18	6	96	97
ligne :	96	97	-18	8	-1	97
ligne :	97	98	-53	94	-1	98
ligne :	98	99	-51	97	103	104
ligne :	99	100	-45	-1	100	101
ligne :	100	101	-54	2	-1	101
ligne :	101	102	-93	99	-1	102
ligne :	102	103	-40	101	-1	103
ligne :	103	104	-35	102	-1	104
ligne :	104	105	-83	98	113	176
ligne :	105	106	-18	4	106	109
ligne :	106	107	-18	6	107	109
ligne :	107	108	-18	9	108	109
ligne :	108	109	-18	10	-1	109
ligne :	109	110	-53	105	-1	110
ligne :	110	111	-51	109	112	113
ligne :	111	112	-56	3	-1	112
ligne :	112	113	-35	111	-1	113
ligne :	113	114	-83	110	123	176
ligne :	114	115	-18	4	115	118
ligne :	115	116	-18	6	116	118
ligne :	116	117	-18	9	117	118
ligne :	117	118	-18	11	-1	118
ligne :	118	119	-53	114	-1	119
ligne :	119	120	-51	118	122	123
ligne :	120	121	-54	4	-1	121
ligne :	121	122	-40	120	-1	122

line number		ident	type	f_son	brother	father
ligne :	122	123	-35	121	-1	123
ligne :	123	124	-83	119	133	176
ligne :	124	125	-18	4	125	128
ligne :	125	126	-18	6	126	128
ligne :	126	127	-18	9	127	128
ligne :	127	128	-18	12	-1	128
ligne :	128	129	-53	124	-1	129
ligne :	129	130	-51	128	132	133
ligne :	130	131	-54	5	-1	131
ligne :	131	132	-40	130	-1	132
ligne :	132	133	-35	131	-1	133
ligne :	133	134	-83	129	159	176
ligne :	134	135	-18	4	135	138
ligne :	135	136	-18	6	136	138
ligne :	136	137	-18	14	137	138
ligne :	137	138	-18	16	-1	138
ligne :	138	139	-53	134	-1	139
ligne :	139	140	-51	138	158	159
ligne :	140	141	-54	6	-1	141
ligne :	141	142	-40	140	-1	142
ligne :	142	143	-35	141	145	155
ligne :	143	144	-54	7	-1	144
ligne :	144	145	-40	143	-1	145
ligne :	145	146	-35	144	148	155
ligne :	146	147	-54	8	-1	147
ligne :	147	148	-40	146	-1	148
ligne :	148	149	-35	147	151	155
ligne :	149	150	-54	9	-1	150
ligne :	150	151	-40	149	-1	151
ligne :	151	152	-35	150	154	155
ligne :	152	153	-54	10	-1	153
ligne :	153	154	-40	152	-1	154
ligne :	154	155	-35	153	-1	155
ligne :	155	156	-95	142	-1	156
ligne :	156	157	-94	155	-1	157
ligne :	157	158	-40	156	-1	158
ligne :	158	159	-35	157	-1	159
ligne :	159	160	-83	139	166	176
ligne :	160	161	-18	4	161	162
ligne :	161	162	-18	17	-1	162
ligne :	162	163	-53	160	-1	163
ligne :	163	164	-51	162	165	166
ligne :	164	165	-56	11	-1	165
ligne :	165	166	-35	164	-1	166
ligne :	166	167	-83	163	175	176
ligne :	167	168	-18	19	-1	168
ligne :	168	169	-51	167	174	175
ligne :	169	170	-18	4	170	171
ligne :	170	171	-18	5	-1	171
ligne :	171	172	-53	169	-1	172
ligne :	172	173	-51	171	-1	173
ligne :	173	174	-40	172	-1	174
ligne :	174	175	-35	173	-1	175
ligne :	175	176	-83	168	-1	176
ligne :	176	177	-65	84	-1	177
ligne :	177	178	-25	5	-1	178
ligne :	178	179	-5	1	-1	-1

2) The symbol table.

line number		length	class	level	tree	value
ligne :	0	4	-2	1	0	alg1
ligne :	1	1	-5	1	6	a
ligne :	2	1	-5	1	7	b
ligne :	3	1	-5	1	8	c
ligne :	4	1	-5	1	12	d
ligne :	5	4	-5	2	14	test
ligne :	6	2	-5	2	18	d1
ligne :	7	4	-5	3	20	test
ligne :	8	3	-5	3	24	d11
ligne :	9	3	-5	3	28	d12
ligne :	10	4	-5	4	30	test
ligne :	11	4	-5	4	34	d121
ligne :	12	4	-5	4	38	d122
ligne :	13	3	-5	3	44	d13
ligne :	14	3	-5	3	48	d14
ligne :	15	4	-5	4	50	test
ligne :	16	4	-5	4	55	d141
ligne :	17	2	-5	2	64	d2
ligne :	18	1	-5	1	70	g
ligne :	19	1	-5	1	71	h
ligne :	20	1	-5	1	72	i

3) The constant table.

line number		length	value
ligne :	0	5	hello
ligne :	1	1	3
ligne :	2	2	3.
ligne :	3	4	true
ligne :	4	1	4
ligne :	5	2	4.
ligne :	6	1	1
ligne :	7	1	2
ligne :	8	1	4
ligne :	9	1	6
ligne :	10	1	8
ligne :	11	5	false

4) The parse tree.

```
t_alg 179 INCONNU 0
! t_alh 2 t_alg 179
! ! t_ident 1 t_alh 2
! ! ! alg1 0 t_ident 1
! t_alb 178 t_alg 179
! ! t_exdp 6 t_alb 178
! ! ! t_schdec 3 t_exdp 6
! ! ! t_algdec 4 t_exdp 6
! ! ! t_funcdec 5 t_exdp 6
! ! t_vadp 77 t_alb 178
! ! ! t_vardec 12 t_vadp 77
! ! ! ! t_names 10 t_vardec 12
! ! ! ! ! t_ident 7 t_names 10
! ! ! ! ! ! a 1 t_ident 7
! ! ! ! ! t_ident 8 t_names 10
! ! ! ! ! ! b 2 t_ident 8
! ! ! ! ! t_ident 9 t_names 10
! ! ! ! ! ! c 3 t_ident 9
! ! ! ! t_real 11 t_vardec 12
```

```
! ! ! t_vardec 70 t_vadp 77
! ! ! ! t_names 14 t_vardec 70
! ! ! ! ! t_ident 13 t_names 14
! ! ! ! ! ! d 4 t_ident 13
! ! ! ! t_struc 69 t_vardec 70
! ! ! ! ! t_vardec 18 t_struc 69
! ! ! ! ! ! t_names 16 t_vardec 18
! ! ! ! ! ! ! t_ident 15 t_names 16
! ! ! ! ! ! ! ! test 5 t_ident 15
! ! ! ! ! ! t_str 17 t_vardec 18
! ! ! ! ! t_vardec 64 t_struc 69
! ! ! ! ! ! t_names 20 t_vardec 64
! ! ! ! ! ! ! t_ident 19 t_names 20
! ! ! ! ! ! ! ! d1 6 t_ident 19
! ! ! ! ! ! t_struc 63 t_vardec 64
! ! ! ! ! ! ! t_vardec 24 t_struc 63
! ! ! ! ! ! ! ! t_names 22 t_vardec 24
! ! ! ! ! ! ! ! ! t_ident 21 t_names 22
! ! ! ! ! ! ! ! ! ! test 7 t_ident 21
! ! ! ! ! ! ! ! t_int 23 t_vardec 24
! ! ! ! ! ! ! t_vardec 28 t_struc 63
! ! ! ! ! ! ! ! t_names 26 t_vardec 28
! ! ! ! ! ! ! ! ! t_ident 25 t_names 26
! ! ! ! ! ! ! ! ! ! d11 8 t_ident 25
! ! ! ! ! ! ! ! t_real 27 t_vardec 28
! ! ! ! ! ! ! t_vardec 44 t_struc 63
! ! ! ! ! ! ! ! t_names 30 t_vardec 44
! ! ! ! ! ! ! ! ! t_ident 29 t_names 30
! ! ! ! ! ! ! ! ! ! d12 9 t_ident 29
! ! ! ! ! ! ! ! t_struc 43 t_vardec 44
! ! ! ! ! ! ! ! ! t_vardec 34 t_struc 43
! ! ! ! ! ! ! ! ! ! t_names 32 t_vardec 34
! ! ! ! ! ! ! ! ! ! ! t_ident 31 t_names 32
! ! ! ! ! ! ! ! ! ! ! ! test 10 t_ident 31
! ! ! ! ! ! ! ! ! ! t_bool 33 t_vardec 34
! ! ! ! ! ! ! ! ! t_vardec 38 t_struc 43
! ! ! ! ! ! ! ! ! ! t_names 36 t_vardec 38
! ! ! ! ! ! ! ! ! ! ! t_ident 35 t_names 36
! ! ! ! ! ! ! ! ! ! ! ! d121 11 t_ident 35
! ! ! ! ! ! ! ! ! ! t_int 37 t_vardec 38
! ! ! ! ! ! ! ! ! t_vardec 42 t_struc 43
! ! ! ! ! ! ! ! ! ! t_names 40 t_vardec 42
! ! ! ! ! ! ! ! ! ! ! t_ident 39 t_names 40
! ! ! ! ! ! ! ! ! ! ! ! d122 12 t_ident 39
! ! ! ! ! ! ! ! ! ! t_real 41 t_vardec 42
```

```
! ! ! ! ! ! ! t_vardec 48 t_struc 63
! ! ! ! ! ! ! ! t_names 46 t_vardec 48
! ! ! ! ! ! ! ! ! t_ident 45 t_names 46
! ! ! ! ! ! ! ! ! ! d13 13 t_ident 45
! ! ! ! ! ! ! ! t_bool 47 t_vardec 48
! ! ! ! ! ! ! t_vardec 62 t_struc 63
! ! ! ! ! ! ! ! t_names 50 t_vardec 62
! ! ! ! ! ! ! ! ! t_ident 49 t_names 50
! ! ! ! ! ! ! ! ! ! d14 14 t_ident 49
! ! ! ! ! ! ! ! t_struc 61 t_vardec 62
! ! ! ! ! ! ! ! ! t_vardec 55 t_struc 61
! ! ! ! ! ! ! ! ! ! t_names 52 t_vardec 55
! ! ! ! ! ! ! ! ! ! ! t_ident 51 t_names 52
! ! ! ! ! ! ! ! ! ! ! ! test 15 t_ident 51
! ! ! ! ! ! ! ! ! ! t_listyp 54 t_vardec 55
! ! ! ! ! ! ! ! ! ! ! t_str 53 t_listyp 54
! ! ! ! ! ! ! ! ! t_vardec 60 t_struc 61
! ! ! ! ! ! ! ! ! ! t_names 57 t_vardec 60
! ! ! ! ! ! ! ! ! ! ! t_ident 56 t_names 57
! ! ! ! ! ! ! ! ! ! ! ! d141 16 t_ident 56
! ! ! ! ! ! ! ! ! ! t_listyp 59 t_vardec 60
! ! ! ! ! ! ! ! ! ! ! t_int 58 t_listyp 59
! ! ! ! ! t_vardec 68 t_struc 69
! ! ! ! ! ! t_names 66 t_vardec 68
! ! ! ! ! ! ! t_ident 65 t_names 66
! ! ! ! ! ! ! ! d2 17 t_ident 65
! ! ! ! ! ! t_bool 67 t_vardec 68
! ! ! t_vardec 76 t_vadp 77
! ! ! ! t_names 74 t_vardec 76
! ! ! ! ! t_ident 71 t_names 74
! ! ! ! ! ! g 18 t_ident 71
! ! ! ! ! t_ident 72 t_names 74
! ! ! ! ! ! h 19 t_ident 72
! ! ! ! ! t_ident 73 t_names 74
! ! ! ! ! ! i 20 t_ident 73
! ! ! ! t_str 75 t_vardec 76
```

```
! ! t_stp 177 t_alb 178
! ! ! t_assst 85 t_stp 177
! ! ! ! t_var 81 t_assst 85
! ! ! ! ! t_struct 80 t_var 81
! ! ! ! ! ! t_ident 78 t_struct 80
! ! ! ! ! ! ! d 4 t_ident 78
! ! ! ! ! ! t_ident 79 t_struct 80
! ! ! ! ! ! ! test 5 t_ident 79
! ! ! ! t_genexp 84 t_assst 85
! ! ! ! ! t_arexp 83 t_genexp 84
! ! ! ! ! ! t_str_k 82 t_arexp 83
! ! ! ! ! ! ! hello 0 t_str_k 82
! ! ! t_assst 94 t_stp 177
! ! ! ! t_var 90 t_assst 94
! ! ! ! ! t_struct 89 t_var 90
! ! ! ! ! ! t_ident 86 t_struct 89
! ! ! ! ! ! ! d 4 t_ident 86
! ! ! ! ! ! t_ident 87 t_struct 89
! ! ! ! ! ! ! d1 6 t_ident 87
! ! ! ! ! ! t_ident 88 t_struct 89
! ! ! ! ! ! ! test 7 t_ident 88
! ! ! ! t_genexp 93 t_assst 94
! ! ! ! ! t_arexp 92 t_genexp 93
! ! ! ! ! ! t_unsnum 91 t_arexp 92
! ! ! ! ! ! ! 3 1 t_unsnum 91
! ! ! t_assst 105 t_stp 177
! ! ! ! t_var 99 t_assst 105
! ! ! ! ! t_struct 98 t_var 99
! ! ! ! ! ! t_ident 95 t_struct 98
! ! ! ! ! ! ! d 4 t_ident 95
! ! ! ! ! ! t_ident 96 t_struct 98
! ! ! ! ! ! ! d1 6 t_ident 96
! ! ! ! ! ! t_ident 97 t_struct 98
! ! ! ! ! ! ! d11 8 t_ident 97
! ! ! ! t_genexp 104 t_assst 105
! ! ! ! ! t_arexp 103 t_genexp 104
! ! ! ! ! ! t_adexpu 102 t_arexp 103
! ! ! ! ! ! ! t_minus 100 t_adexpu 102
! ! ! ! ! ! ! t_unsnum 101 t_adexpu 102
! ! ! ! ! ! ! ! 3. 2 t_unsnum 101
! ! ! t_assst 114 t_stp 177
! ! ! ! t_var 111 t_assst 114
! ! ! ! ! t_struct 110 t_var 111
! ! ! ! ! ! t_ident 106 t_struct 110
! ! ! ! ! ! ! d 4 t_ident 106
! ! ! ! ! ! t_ident 107 t_struct 110
! ! ! ! ! ! ! d1 6 t_ident 107
! ! ! ! ! ! t_ident 108 t_struct 110
! ! ! ! ! ! ! d12 9 t_ident 108
! ! ! ! ! ! t_ident 109 t_struct 110
! ! ! ! ! ! ! test 10 t_ident 109
! ! ! ! t_genexp 113 t_assst 114
! ! ! ! ! t_logval 112 t_genexp 113
! ! ! ! ! ! true 3 t_logval 112
```

```
! ! ! t_assst 124 t_stp 177
! ! ! ! t_var 120 t_assst 124
! ! ! ! ! t_struct 119 t_var 120
! ! ! ! ! ! t_ident 115 t_struct 119
! ! ! ! ! ! ! d 4 t_ident 115
! ! ! ! ! ! t_ident 116 t_struct 119
! ! ! ! ! ! ! d1 6 t_ident 116
! ! ! ! ! ! t_ident 117 t_struct 119
! ! ! ! ! ! ! d12 9 t_ident 117
! ! ! ! ! ! t_ident 118 t_struct 119
! ! ! ! ! ! ! d121 11 t_ident 118
! ! ! ! t_genexp 123 t_assst 124
! ! ! ! ! t_arexp 122 t_genexp 123
! ! ! ! ! ! t_unsnum 121 t_arexp 122
! ! ! ! ! ! ! 4 4 t_unsnum 121
! ! ! t_assst 134 t_stp 177
! ! ! ! t_var 130 t_assst 134
! ! ! ! ! t_struct 129 t_var 130
! ! ! ! ! ! t_ident 125 t_struct 129
! ! ! ! ! ! ! d 4 t_ident 125
! ! ! ! ! ! t_ident 126 t_struct 129
! ! ! ! ! ! ! d1 6 t_ident 126
! ! ! ! ! ! t_ident 127 t_struct 129
! ! ! ! ! ! ! d12 9 t_ident 127
! ! ! ! ! ! t_ident 128 t_struct 129
! ! ! ! ! ! ! d122 12 t_ident 128
! ! ! ! t_genexp 133 t_assst 134
! ! ! ! ! t_arexp 132 t_genexp 133
! ! ! ! ! ! t_unsnum 131 t_arexp 132
! ! ! ! ! ! ! 4. 5 t_unsnum 131
! ! ! t_assst 160 t_stp 177
! ! ! ! t_var 140 t_assst 160
! ! ! ! ! t_struct 139 t_var 140
! ! ! ! ! ! t_ident 135 t_struct 139
! ! ! ! ! ! ! d 4 t_ident 135
! ! ! ! ! ! t_ident 136 t_struct 139
! ! ! ! ! ! ! d1 6 t_ident 136
! ! ! ! ! ! t_ident 137 t_struct 139
! ! ! ! ! ! ! d14 14 t_ident 137
! ! ! ! ! ! t_ident 138 t_struct 139
! ! ! ! ! ! ! d141 16 t_ident 138
```

```
! ! ! ! t_genexp 159 t_assst 160
! ! ! ! ! t_arexp 158 t_genexp 159
! ! ! ! ! ! t_collexp 157 t_arexp 158
! ! ! ! ! ! ! t_list 156 t_collexp 157
! ! ! ! ! ! ! ! t_genexp 143 t_list 156
! ! ! ! ! ! ! ! ! t_arexp 142 t_genexp 143
! ! ! ! ! ! ! ! ! ! t_unsnum 141 t_arexp 142
! ! ! ! ! ! ! ! ! ! ! 1 6 t_unsnum 141
! ! ! ! ! ! ! ! t_genexp 146 t_list 156
! ! ! ! ! ! ! ! ! t_arexp 145 t_genexp 146
! ! ! ! ! ! ! ! ! ! t_unsnum 144 t_arexp 145
! ! ! ! ! ! ! ! ! ! ! 2 7 t_unsnum 144
! ! ! ! ! ! ! ! t_genexp 149 t_list 156
! ! ! ! ! ! ! ! ! t_arexp 148 t_genexp 149
! ! ! ! ! ! ! ! ! ! t_unsnum 147 t_arexp 148
! ! ! ! ! ! ! ! ! ! ! 4 8 t_unsnum 147
! ! ! ! ! ! ! ! t_genexp 152 t_list 156
! ! ! ! ! ! ! ! ! t_arexp 151 t_genexp 152
! ! ! ! ! ! ! ! ! ! t_unsnum 150 t_arexp 151
! ! ! ! ! ! ! ! ! ! ! 6 9 t_unsnum 150
! ! ! ! ! ! ! ! t_genexp 155 t_list 156
! ! ! ! ! ! ! ! ! t_arexp 154 t_genexp 155
! ! ! ! ! ! ! ! ! ! t_unsnum 153 t_arexp 154
! ! ! ! ! ! ! ! ! ! ! 8 10 t_unsnum 153
! ! ! t_assst 167 t_stp 177
! ! ! ! t_var 164 t_assst 167
! ! ! ! ! t_struct 163 t_var 164
! ! ! ! ! ! t_ident 161 t_struct 163
! ! ! ! ! ! ! d 4 t_ident 161
! ! ! ! ! ! t_ident 162 t_struct 163
! ! ! ! ! ! ! d2 17 t_ident 162
! ! ! ! t_genexp 166 t_assst 167
! ! ! ! ! t_logval 165 t_genexp 166
! ! ! ! ! ! false 11 t_logval 165
! ! ! t_assst 176 t_stp 177
! ! ! ! t_var 169 t_assst 176
! ! ! ! ! t_ident 168 t_var 169
! ! ! ! ! ! h 19 t_ident 168
! ! ! ! t_genexp 175 t_assst 176
! ! ! ! ! t_arexp 174 t_genexp 175
! ! ! ! ! ! t_var 173 t_arexp 174
! ! ! ! ! ! ! t_struct 172 t_var 173
! ! ! ! ! ! ! ! t_ident 170 t_struct 172
! ! ! ! ! ! ! ! ! d 4 t_ident 170
! ! ! ! ! ! ! ! t_ident 171 t_struct 172
! ! ! ! ! ! ! ! ! test 5 t_ident 171
```

```
algorithm collection
schema s1
funcs print : integer
var CUS:record CUSTOMER
begin
for CUS:=CUSTOMER(CUSOR: [1:*] ORDER) and (:NAME = 'smith') do
        print(NAME(:CUS))
endfor
end
```

1) The node table

line number		ident	type	f_son	brother	father
ligne :	0	1	-18	0	-1	1
ligne :	1	2	-6	0	63	64
ligne :	2	3	-18	1	-1	3
ligne :	3	4	-88	2	4	9
ligne :	4	5	-11	-1	8	9
ligne :	5	6	-18	15	6	7
ligne :	6	7	-22	-1	-1	7
ligne :	7	8	-13	5	-1	8
ligne :	8	9	-12	7	-1	9
ligne :	9	10	-7	3	15	63
ligne :	10	11	-18	16	-1	11
ligne :	11	12	-27	10	13	14
ligne :	12	13	-18	2	-1	13
ligne :	13	14	-23	12	-1	14
ligne :	14	15	-17	11	-1	15
ligne :	15	16	-16	14	62	63
ligne :	16	17	-18	16	-1	17
ligne :	17	18	-51	16	45	61
ligne :	18	19	-18	2	40	41
ligne :	19	20	-18	9	-1	25
ligne :	20	21	-54	0	-1	21
ligne :	21	22	-40	20	-1	22
ligne :	22	23	-35	21	-1	23
ligne :	23	24	-103	22	-1	24
ligne :	24	25	-60	23	19	25
ligne :	25	26	-49	24	27	28
ligne :	26	27	-18	3	-1	27
ligne :	27	28	-51	26	-1	28
ligne :	28	29	-63	25	38	39
ligne :	29	30	-57	-1	-1	30
ligne :	30	31	-49	29	37	38
ligne :	31	32	-18	6	36	37
ligne :	32	33	-58	1	34	35
ligne :	33	34	-2	2	-1	34
ligne :	34	35	-40	33	-1	-1

line number		ident	type	f_son	brother	father
ligne :	35	36	-50	32	-1	36
ligne :	36	37	-97	35	-1	37
ligne :	37	38	-96	31	-1	38
ligne :	38	39	-63	30	-1	39
ligne :	39	40	-99	28	-1	40
ligne :	40	41	-97	39	-1	41
ligne :	41	42	-96	18	-1	42
ligne :	42	43	-94	41	-1	43
ligne :	43	44	-40	42	-1	44
ligne :	44	45	-35	43	-1	45
ligne :	45	46	-67	44	60	61
ligne :	46	47	-18	15	58	59
ligne :	47	48	-18	6	53	54
ligne :	48	49	-57	-1	-1	49
ligne :	49	50	-49	48	51	52
ligne :	50	51	-18	16	-1	51
ligne :	51	52	-51	50	-1	52
ligne :	52	53	-63	49	-1	53
ligne :	53	54	-97	52	-1	54
ligne :	54	55	-96	47	-1	55
ligne :	55	56	-94	54	-1	56
ligne :	56	57	-40	55	-1	57
ligne :	57	58	-35	56	-1	58
ligne :	58	59	-87	57	-1	59
ligne :	59	60	-82	46	-1	60
ligne :	60	61	-90	59	-1	61
ligne :	61	62	-66	17	-1	62
ligne :	62	63	-65	61	-1	63
ligne :	63	64	-25	9	-1	64
ligne :	64	65	-5	1	-1	-1

2) The symbol table.

line number		length	class	level	tree	value
ligne :	0	10	-2	1	0	collection
ligne :	1	2	-1	1	2	s1
ligne :	2	8	-3	1	-1	CUSTOMER
ligne :	3	5	-3	1	-1	ORDER
ligne :	4	10	-3	1	-1	ORDER_LINE
ligne :	5	7	-3	1	-1	PRODUCT
ligne :	6	4	-6	1	-1	NAME
ligne :	7	3	-6	1	-1	NUM
ligne :	8	8	-6	1	-1	QUANTITY
ligne :	9	5	-7	1	-1	CUSOR
ligne :	10	5	-7	1	-1	ORCUS
ligne :	11	4	-7	1	-1	ORLI
ligne :	12	4	-7	1	-1	LIOR
ligne :	13	5	-7	1	-1	LIPRO
ligne :	14	5	-7	1	-1	PROLI
ligne :	15	5	-4	1	5	print
ligne :	16	3	-5	1	10	CUS

3) The constant table.

line number		length	value
ligne :	0	1	1
ligne :	1	1	=
ligne :	2	5	smith

4) The parse tree.

```
t_alg 65 INCONNU 0
! t_alh 2 t_alg 65
! ! t_ident 1 t_alh 2
! ! ! collection 0 t_ident 1
! t_alb 64 t_alg 65
! ! t_exdp 10 t_alb 64
! ! ! t_schdec 4 t_exdp 10
! ! ! ! t_ident 3 t_schdec 4
! ! ! ! ! s1 1 t_ident 3
! ! ! t_algdec 5 t_exdp 10
! ! ! t_funcdec 9 t_exdp 10
! ! ! ! t_felt 8 t_funcdec 9
! ! ! ! ! t_ident 6 t_felt 8
! ! ! ! ! ! print 15 t_ident 6
! ! ! ! ! t_int 7 t_felt 8
! ! t_vadp 16 t_alb 64
! ! ! t_vardec 15 t_vadp 16
! ! ! ! t_names 12 t_vardec 15
! ! ! ! ! t_ident 11 t_names 12
! ! ! ! ! ! CUS 16 t_ident 11
! ! ! ! t_rectyp 14 t_vardec 15
! ! ! ! ! t_ident 13 t_rectyp 14
! ! ! ! ! ! CUSTOMER 2 t_ident 13
! ! t_stp 63 t_alb 64
! ! ! t_enumloop 62 t_stp 63
! ! ! ! t_var 18 t_enumloop 62
! ! ! ! ! t_ident 17 t_var 18
! ! ! ! ! ! CUS 16 t_ident 17
! ! ! ! t_forlis 46 t_enumloop 62
! ! ! ! ! t_genexp 45 t_forlis 46
! ! ! ! ! ! t_arexp 44 t_genexp 45
! ! ! ! ! ! ! t_collexp 43 t_arexp 44
! ! ! ! ! ! ! ! t_dbset 42 t_collexp 43
! ! ! ! ! ! ! ! ! t_ident 19 t_dbset 42
! ! ! ! ! ! ! ! ! ! CUSTOMER 2 t_ident 19
! ! ! ! ! ! ! ! ! t_predicate 41 t_dbset 42
! ! ! ! ! ! ! ! ! ! t_ccfac 40 t_predicate 41
! ! ! ! ! ! ! ! ! ! ! t_relcon 29 t_ccfac 40
! ! ! ! ! ! ! ! ! ! ! ! t_relop 26 t_relcon 29
! ! ! ! ! ! ! ! ! ! ! ! ! t_card 25 t_relop 26
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_rangel 24 t_card 25
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_genexp 23 t_rangel 24
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_arexp 22 t_genexp 23
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_unsnum 21 t_arexp 22
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! 1 0 t_unsnum 21
! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 20 t_relop 26
! ! ! ! ! ! ! ! ! ! ! ! ! ! CUSOR 9 t_ident 20
! ! ! ! ! ! ! ! ! ! ! ! t_var 28 t_relcon 29
! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 27 t_var 28
! ! ! ! ! ! ! ! ! ! ! ! ! ! ORDER 3 t_ident 27
! ! ! ! ! ! ! ! ! ! ! t_relcon 39 t_ccfac 40
! ! ! ! ! ! ! ! ! ! ! ! t_relop 31 t_relcon 39
! ! ! ! ! ! ! ! ! ! ! ! ! t_co 30 t_relop 31
! ! ! ! ! ! ! ! ! ! ! ! t_dbset 38 t_relcon 39
! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 32 t_dbset 38
! ! ! ! ! ! ! ! ! ! ! ! ! ! NAME 6 t_ident 32
! ! ! ! ! ! ! ! ! ! ! ! ! t_predicate 37 t_dbset 38
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_belcon 36 t_predicate 37
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_tesop 33 t_belcon 36
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! = 1 t_tesop 33
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_arexp 35 t_bel_con 36
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_str_k 34 t_arexp 35
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! smith 2 t_str_k 34
```

```
! ! ! ! t_actions 61 t_enumloop 62
! ! ! ! ! t_call 60 t_actions 61
! ! ! ! ! ! t_ident 47 t_call 60
! ! ! ! ! ! ! print 15 t_ident 47
! ! ! ! ! ! t_parlis 59 t_call 60
! ! ! ! ! ! ! t_genexp 58 t_parlis 59
! ! ! ! ! ! ! ! t_arexp 57 t_genexp 58
! ! ! ! ! ! ! ! ! t_collexp 56 t_arexp 57
! ! ! ! ! ! ! ! ! ! t_dbset 55 t_collexp 56
! ! ! ! ! ! ! ! ! ! ! t_ident 48 t_dbset 55
! ! ! ! ! ! ! ! ! ! ! ! NAME 6 t_ident 48
! ! ! ! ! ! ! ! ! ! ! t_predicate 54 t_dbset 55
! ! ! ! ! ! ! ! ! ! ! ! t_relcon 53 t_predicate 54
! ! ! ! ! ! ! ! ! ! ! ! ! t_relop 50 t_relcon 53
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_co 49 t_relop 50
! ! ! ! ! ! ! ! ! ! ! ! ! t_var 52 t_relcon 53
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 51 t_var 52
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! CUS 16 t_ident 51
```

```
algorithm predicativ1

schema  example1

funcs  print : integer

var CUS : record CUSTOMER

begin

for CUS := CUSTOMER(CUSOR : ORDER(ORLI : ORDER_LINE(LIPRO:

               PRODUCT(: NUM = 783)))) do

               print(NAME(:CUS))

endfor
end
```

1) The node table.

line number		ident	type	f_son	brother	father
ligne :	0	1	-18	0	-1	1
ligne :	1	2	-6	0	73	74
ligne :	2	3	-18	1	-1	3
ligne :	3	4	-88	2	4	9
ligne :	4	5	-11	-1	8	9
ligne :	5	6	-18	15	6	7
ligne :	6	7	-22	-1	-1	7
ligne :	7	8	-13	5	-1	8
ligne :	8	9	-12	7	-1	9
ligne :	9	10	-7	3	15	73
ligne :	10	11	-18	16	-1	11
ligne :	11	12	-27	10	13	14
ligne :	12	13	-18	2	-1	13
ligne :	13	14	-23	12	-1	14
ligne :	14	15	-17	11	-1	15
ligne :	15	16	-16	14	72	73
ligne :	16	17	-18	16	-1	17
ligne :	17	18	-51	16	55	71
ligne :	18	19	-18	2	50	51
ligne :	19	20	-18	9	-1	21
ligne :	20	21	-57	-1	19	21
ligne :	21	22	-49	20	48	49
ligne :	22	23	-18	3	47	48
ligne :	23	24	-18	11	-1	25
ligne :	24	25	-57	-1	23	25
ligne :	25	26	-49	24	45	46
ligne :	26	27	-18	4	44	45
ligne :	27	28	-18	13	-1	29
ligne :	28	29	-57	-1	27	29
ligne :	29	30	-49	28	42	43
ligne :	30	31	-18	5	41	42
ligne :	31	32	-57	-1	-1	32
ligne :	32	33	-49	31	39	40
ligne :	33	34	-18	7	38	39

line number		ident	type	f_son	brother	father
ligne :	34	35	-58	0	36	37
ligne :	35	36	-54	1	-1	36
ligne :	36	37	-40	35	-1	-1
ligne :	37	38	-50	34	-1	38
ligne :	38	39	-97	37	-1	39
ligne :	39	40	-96	33	-1	40
ligne :	40	41	-63	32	-1	41
ligne :	41	42	-97	40	-1	42
ligne :	42	43	-96	30	-1	43
ligne :	43	44	-63	29	-1	44
ligne :	44	45	-97	43	-1	45
ligne :	45	46	-96	26	-1	46
ligne :	46	47	-63	25	-1	47
ligne :	47	48	-97	46	-1	48
ligne :	48	49	-96	22	-1	49
ligne :	49	50	-63	21	-1	50
ligne :	50	51	-97	49	-1	51
ligne :	51	52	-96	18	-1	52
ligne :	52	53	-94	51	-1	53
ligne :	53	54	-40	52	-1	54
ligne :	54	55	-35	53	-1	55
ligne :	55	56	-67	54	70	71
ligne :	56	57	-18	15	68	69
ligne :	57	58	-18	6	63	64
ligne :	58	59	-57	-1	-1	59
ligne :	59	60	-49	58	61	62
ligne :	60	61	-18	16	-1	61
ligne :	61	62	-51	60	-1	62
ligne :	62	63	-63	59	-1	63
ligne :	63	64	-97	62	-1	64
ligne :	64	65	-96	57	-1	65
ligne :	65	66	-94	64	-1	66
ligne :	66	67	-40	65	-1	67
ligne :	67	68	-35	66	-1	68
ligne :	68	69	-87	67	-1	69
ligne :	69	70	-82	56	-1	70
ligne :	70	71	-90	69	-1	71
ligne :	71	72	-66	17	-1	72
ligne :	72	73	-65	71	-1	73
ligne :	73	74	-25	9	-1	74
ligne :	74	75	-5	1	-1	-1

2) The symbol table.

line number		length	class	level	tree	value
ligne :	0	11	-2	1	0	predicativ1
ligne :	1	8	-1	1	2	example1
ligne :	2	8	-3	1	-1	CUSTOMER
ligne :	3	5	-3	1	-1	ORDER
ligne :	4	10	-3	1	-1	ORDER_LINE
ligne :	5	7	-3	1	-1	PRODUCT
ligne :	6	4	-6	1	-1	NAME
ligne :	7	3	-6	1	-1	NUM
ligne :	8	8	-6	1	-1	QUANTITY
ligne :	9	5	-7	1	-1	CUSOR
ligne :	10	5	-7	1	-1	ORCUS
ligne :	11	4	-7	1	-1	ORLI
ligne :	12	4	-7	1	-1	LIOR
ligne :	13	5	-7	1	-1	LIPRO
ligne :	14	5	-7	1	-1	PROLI
ligne :	15	5	-4	1	5	print
ligne :	16	3	-5	1	10	CUS

3) The constant table.

line number		length	value
ligne :	0	1	=
ligne :	1	3	783

4) The parse tree.

```
t_alg 75 INCONNU 0
! t_alh 2 t_alg 75
! ! t_ident 1 t_alh 2
! ! ! predicativ1 0 t_ident 1
! t_alb 74 t_alg 75
! ! t_exdp 10 t_alb 74
! ! ! t_schdec 4 t_exdp 10
! ! ! ! t_ident 3 t_schdec 4
! ! ! ! ! example1 1 t_ident 3
! ! ! t_algdec 5 t_exdp 10
! ! ! t_funcdec 9 t_exdp 10
! ! ! ! t_felt 8 t_funcdec 9
! ! ! ! ! t_ident 6 t_felt 8
! ! ! ! ! ! print 15 t_ident 6
! ! ! ! ! t_int 7 t_felt 8
! ! t_vadp 16 t_alb 74
! ! ! t_vardec 15 t_vadp 16
! ! ! ! t_names 12 t_vardec 15
! ! ! ! ! t_ident 11 t_names 12
! ! ! ! ! ! CUS 16 t_ident 11
! ! ! ! t_rectyp 14 t_vardec 15
! ! ! ! ! t_ident 13 t_rectyp 14
! ! ! ! ! ! CUSTOMER 2 t_ident 13
```

```
! ! t_stp 73 t_alb 74
! ! ! t_enumloop 72 t_stp 73
! ! ! ! t_var 18 t_enumloop 72
! ! ! ! ! t_ident 17 t_var 18
! ! ! ! ! ! CUS 16 t_ident 17
! ! ! ! t_forlis 56 t_enumloop 72
! ! ! ! ! t_genexp 55 t_forlis 56
! ! ! ! ! ! t_arexp 54 t_genexp 55
! ! ! ! ! ! ! t_collexp 53 t_arexp 54
! ! ! ! ! ! ! ! t_dbset 52 t_collexp 53
! ! ! ! ! ! ! ! ! t_ident 19 t_dbset 52
! ! ! ! ! ! ! ! ! ! CUSTOMER 2 t_ident 19
! ! ! ! ! ! ! ! ! t_predicate 51 t_dbset 52
! ! ! ! ! ! ! ! ! ! t_relcon 50 t_predicate 51
! ! ! ! ! ! ! ! ! ! ! t_relop 22 t_relcon 50
! ! ! ! ! ! ! ! ! ! ! ! t_co 21 t_relop 22
! ! ! ! ! ! ! ! ! ! ! ! t_ident 20 t_relop 22
! ! ! ! ! ! ! ! ! ! ! ! ! CUSOR 9 t_ident 20
! ! ! ! ! ! ! ! ! ! ! t_dbset 49 t_relcon 50
! ! ! ! ! ! ! ! ! ! ! ! t_ident 23 t_dbset 49
! ! ! ! ! ! ! ! ! ! ! ! ! ORDER 3 t_ident 23
! ! ! ! ! ! ! ! ! ! ! ! t_predicate 48 t_dbset 49
! ! ! ! ! ! ! ! ! ! ! ! ! t_relcon 47 t_predicate 48
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_relop 26 t_relcon 47
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_co 25 t_relop 26
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 24 t_relop 26
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ORLI 11 t_ident 24
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_dbset 46 t_relcon 47
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 27 t_dbset 46
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ORDER_LINE 4 t_ident 27
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_predicate 45 t_dbset 46
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_relcon 44 t_predicate 45
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_relop 30 t_relcon 44
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_co 29 t_relop 30
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 28 t_relop 30
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! LIPRO 13 t_ident 28
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_dbset 43 t_relcon 44
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 31 t_dbset 43
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! PRODUCT 5 t_ident 31
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_predicate 42 t_dbset 43
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_relcon 41 t_predicate 42
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_relop 33 t_relcon 41
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_co 32 t_relop 33
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_dbset 40 t_relcon 41
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 34 t_dbset 40
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! NUM 7 t_ident 34
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_predicate 39 t_dbset 40
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_belcon 38 t_predicate 39
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_tesop 35 t_belcon 38
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! = 0 t_tesop 35
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_arexp 37 t_bel_con 38
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! t_unsnum 36 t_arexp 37
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! 783 1 t_unsnum 36
```

```
! ! ! ! t_actions 71 t_enumloop 72
! ! ! ! ! t_call 70 t_actions 71
! ! ! ! ! ! t_ident 57 t_call 70
! ! ! ! ! ! ! print 15 t_ident 57
! ! ! ! ! ! t_parlis 69 t_call 70
! ! ! ! ! ! ! t_genexp 68 t_parlis 69
! ! ! ! ! ! ! ! t_arexp 67 t_genexp 68
! ! ! ! ! ! ! ! ! t_collexp 66 t_arexp 67
! ! ! ! ! ! ! ! ! ! t_dbset 65 t_collexp 66
! ! ! ! ! ! ! ! ! ! ! t_ident 58 t_dbset 65
! ! ! ! ! ! ! ! ! ! ! ! NAME 6 t_ident 58
! ! ! ! ! ! ! ! ! ! ! t_predicate 64 t_dbset 65
! ! ! ! ! ! ! ! ! ! ! ! t_relcon 63 t_predicate 64
! ! ! ! ! ! ! ! ! ! ! ! ! t_relop 60 t_relcon 63
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_co 59 t_relop 60
! ! ! ! ! ! ! ! ! ! ! ! ! t_var 62 t_relcon 63
! ! ! ! ! ! ! ! ! ! ! ! ! ! t_ident 61 t_var 62
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! CUS 16 t_ident 61
```