SOUR

SYSTENA

INTELLIGENT QUERY SYSTEM

Technical Reference Manual

Version: 2
Revision: 0

Part I 2
1 Context 2
1.1 Document Layout 2
1.2 IQS source files 3

Part II 5
2 IQL Parsing 5
2.1 The iqs.l file (using Lex) 6
2.2 The iqs.y file (using Yacc) 8
2.3 Lexer - Parser communication 17
2.4 Structure of the IQL grammar 18
2.4.1 Assisted Mode IQL sub-grammar issues 18
2.4.2 Batch Mode IQL sub-grammar issues 20
3 Importing the generated code to a Windows DLL 20
3.1 Redirecting the Lexer input 20
3.2 Output in the Lexer and in the Parser 23
3.3 The myfunc.c redefinition file 24
3.4 Importing malloc and realloc 24
3.5 The #define __DOS__ directive 25
3.6 The Large model of compilation 26

Part III 28
4 IQS main data structures (iqs.h) 28
4.1 Handling the repository information (the Set approach) 28
4.1.1 Dynamic Arrays implementing Sets 28
4.1.2 A basic Set API 30
4.2 The *ParserState* data type 33
4.2.1 Implementation details of the History 36
4.2.2 The VBState data type 38
5 The IQS API (iqs.c) 39
5.1 An Auxiliary IQS API 49
6 The IQS Semantic Actions API (actions.c) 51
6.1 The IQS Semantic Actions Auxiliary API 64
7 The IQS Visual Basic related API 66
8 IQS module cross reference 72
8.1 Cross reference for the iqs.c file 72
8.1.1 IQS API Auxiliary functions 73
8.1.2 Set API functions 73
8.1.3 IQS API functions 73
8.1.4 IQS Visual Basic related API functions 75
8.2 Cross reference for the actions.c file 75
8.2.1 IQS Semantic Actions Auxiliary API functions 75
8.2.2 IQS Semantic Actions API functions 76

References 78

Part I

1 Context

This document is the Technical Reference manual of the Intelligent Query Subsystem (IQS) of the SOUR software system.

It describes the most important implementation issues concerning IQS as well as providing enough information for those in charge with maintenance of the IQS module.

This text has a bottom-up style of presentation. It starts by describing certain low-level implementation aspects, and proceeds towards interface issues. However, the interface details covered in this document are only the ones sitting below the Visual Basic/C frontier, that is, only those C implemented functionalities directly callable by Visual Basic will be discussed.

This bottom-up approach is a convenient way of matching the various implementation phases of the IQS module, reflecting many of the decisions taken and even some of the constraints found along the development. Therefore, it serves also as a guideline through the implementation process.

Whenever considered convenient, a little refreshing on some design and architectural aspects will be provided for a better understanding of the implementation decision being described.

IQS *Functional Specifications & Architecture* document [IQS-2.1] should be carefully read in order to better understand many details of this text.

1.1 Document Layout

The first implementation issues to be discussed are those related to the parsing mechanism for the Interface Query Language (IQL)[1]. It seems logical that before starting with the implementation of the software layer in charge with retrieving information from the repository, a way to recognize and handle query sentences be provided.

Once the parsing problems are solved (at least in what concerns to low-level C code; note that interface issues are deferred to later treatment, reflecting our bottom-up approach) it is time to think on the data structures[2] which keep and handle the information retrieved. A set of functions well suited to manipulate each particular data structures will also be presented.

After that, the IQS API is described. The IQS API is a set of functions built around the SOURLIB and exclusively concerned with seeking for the objects on the repository obeying to the present query.

Having the parsing details solved, and the necessary functionalities to access repository, the Semantic Actions can then be presented.

[1]refer to **5 Interface Query Language** on [IQS-2.1].

[2]recall **7 IQS data structure design** on [IQS-2.1].

Some implementation details intimately connected with the interaction Visual Basic Layer/C Layer will then be discussed. In particular, the way in which the Visual Basic Layer and the C Layer exchange information). Again, a set of specialized data structures and functions will be put forward.

Finally, a global cross reference for all the IQS module functions will be presented. This cross reference shows the dependences between the IQS Module functions and any other SOURLIB Modules.

1.2 IQS source files

Before introducing the IQS Software Layers, it is perhaps convenient to offer a brief perspective on the relations between the source files implementing those layers. In a way this disagrees with the above established bottom-up approach, because the tasks these files implement are, for the moment, unknown. On the other hand, it will help to better understand the integration and cooperation of the Software parts of the IQS module.

The below diagram shows the major dependences between the most important files specific to the IQS module. A full arrow means the source file is included in the target file and a dashed one means that the target is generated using the source description. No relation with files from other modules of the SOUR project, as well as with Visual C++ 1.5 libraries is shown.

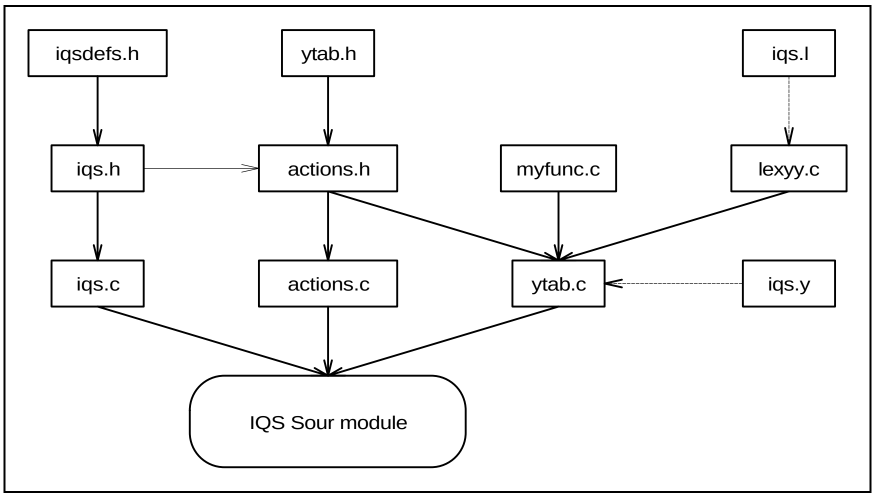

Figure 1 - Relevant dependencies between files of the IQS module

A brief description of each file follows:

- **iqsdefs.h** is a header file defining almost every IQS specific constants;

- **ytab.h** is a header file automatically generated by Yacc; Yacc is a tool that accepts a grammar description and generates C parsing code for that grammar[3];

[3]see **2.2 The iqs.y file (using Yacc)** for more details about this subject.

a full description of **ytab.h** is postponed until section **2.3 Lexer-Parser communication**;

- **iqs.l** is the source file used by Lex to generate **lexyy.c**; Lex is a tool that accepts pattern descriptions and generates C code implementing a scanner for those patterns[4];

- **iqs.h** is the header file of the IQS API[5], defining all the necessary macros, data types, global variables and function prototypes, in order to retrieve information from the repository and exchanging it with the *interface* layer;

- **actions.h** is the header file containing all the definitions needed to implement the semantic actions[6] for the grammar described in **iqs.y**;

- **myfunc.c**[7], contains basic (re)definitions helping the lexical scanning process to work over a string instead of a file;

- **lexyy.c** is the lexical analyzer automatically generated by Lex using the **iqs.l** description;

- **iqs.c** is the file implementing the IQS API.

- **actions.c** contains the semantic actions code for the grammar described in **iqs.y**;

- **ytab.c** is the parser automatically generated by Yacc based on the **iqs.y** description;

- **iqs.y** is the source file used by Yacc to produce **ytab.c** and **ytab.h**.

[4]refer to section **2.1 The iqs.l file (using Lex)**.
[5]see also section **5 The IQS API (iqs.c)** for a complete description.
[6]see section **6 The IQS Semantic Actions API (actions.c)**.
[7]refer to **3.3 The myfunc.c redefinitions file**.

Part II

2 IQL Parsing

During the IQS design phase, two basic operation modes[8] evolved, reflecting two distinct ways of querying the repository:

- the **Assisted Mode**, providing for a permanent syntactic and semantic assistance as well as an exclusively interactive way of query building;

- the **Batch Mode**, less user-friendly but best suited to those users wishing to solve large sets of queries at once and possessing a comprehensive knowledge of the repository structure.

These two operation modes imply two Interface Query Languages[9], designed to cope with the specific demands of each mode, but, at the same time, preserving a minimal degree of compatibility between them.

The formal description of each IQLs is naturally provided by defining its grammar. One advantage of this kind of description is the possibility of using tools allowing for the automatic generation of C code which implements the parsing mechanism for "sentences" obeying to that grammar.

Having two grammars (each one for a specific operation mode, and so for a specific IQL) does not necessarily means of the need for two parsers. It is advantageous to bring together, if possible, both grammars into a single one in order to generate a unique parser: one should not forget that the generated C parsing code must be converted to a WINDOWS 3.1 DLL and therefore it is much easier to deal with only one parser than with two distinct ones, probably sharing many data structures and functions (both parsers would have been generated by the same tool) and rising conflicts hard to solve in automatic generated code environments (to modify generated code can be hard and error prone).

The tools used to generate the C code in charge with syntactical recognition of queries made in both the Assisted and Batch IQL flavors were **Lex & Yacc**. Therefore, the next sections describe the procedure followed in order to have those tools to produce the parsing code for both IQLs (for a matter of convenience we will refer from now on to just one IQL: the one resulting from joining the Batch Mode and the Assisted Mode variants).

Before that, it should be remembered the role of the so-called *Templates*[4] (see Figure 2): these *minimal efficient queries* provide (in design terms) for the basic description of the set of valid tokens Lex must recognize and the set of valid combinations Yacc must deal with. These combinations match already the IQL Batch Mode grammar branch. The IQL Assisted Mode grammar flavor appends to those combinations only the ones enough to deal with the problems of redundancy and incompleteness.

[8]refer to **4 IQS operation modes** from [IQS-2.1].
[9]refer to **5 Interface Query Language** in [IQS-2.1].

```
// Interface Query Language Kernel Templates
NUMBER TEMPLATE1 GET ALL CLASS="class" <AttributeDescription1>*

NUMBER TEMPLATE2 GET ALL CLASS="class" <AttributeDescription2>*

NUMBER TEMPLATE3 GET ALL CLASS="class" <AttributeDescription1>*
                               [ AND IS COMPOUND
                               <CompoundDescription>* ]

NUMBER TEMPLATE4 GET ALL CLASS="class" <AttributeDescription1>*
                               [ AND BELONG TO COMPOUND
                               <CompoundDescription>* ]

// Interface Query Language non-Kernel Templates
NUMBER TEMPLATE5 <Query> [ LINKED BY "relation" [ WITH <Query> ] ]

NUMBER TEMPLATE6 <Query> [ LINKED TO <Query> [ BY "relation" ] ]

NUMBER TEMPLATE7 <Query> [ OR <Query> ]

NUMBER TEMPLATE8 <Query> [ RESTRICTED TO <AttributeDescription1>+ ]

// common definitions
<AttributeDescription1> = AND ( <GenDescp> | <FacDescp> | <AttDescp> )
<GenDescp> = GENNAME="genname" AND GENVALUE="genvalue"
<FacDescp> = FACNAME="facname" AND FACVALUE="facvalue" AND CONCEPTDIST>=<Dist>
<AttDescp> = ATTNAME="attname" AND ATTVALUE="attvalue"

<AttributeDescription2> = ( <AttributeDescription1>* (AND <PhaDescp>)* )*
                          [ AND <SlcDescp> <AttributeDescription1>* ]
<SlcDescp> = SLCNAME="slcname"
<PhaDescp> = PHANAME="phaname"

<CompoundDescription> = <AttributeDescription1>* (AND <CrlDescp>)*
                        <AttributeDescription1>*
<CrlDescp> = CRLNAME="crlname"

<Dist> = NUMBER%
<Query> = #NUMBER

NUMBER = [0-9]+
```

Figure 2 - The Templates

2.1 The iqs.l file (using Lex)

The contents of **iqs.l** describing to Lex the acceptable tokens for the unified IQL grammar Yacc will handle, are:

```
%{
/* definition section */

int yylook();
int yyback(int *, int);

%}
/* some "macros" */

Tpl1              [Tt][Ee][Mm][Pp][Ll][Aa][Tt][Ee][1]
Tpl2              [Tt][Ee][Mm][Pp][Ll][Aa][Tt][Ee][2]
Tpl3              [Tt][Ee][Mm][Pp][Ll][Aa][Tt][Ee][3]
Tpl4              [Tt][Ee][Mm][Pp][Ll][Aa][Tt][Ee][4]
Tpl5              [Tt][Ee][Mm][Pp][Ll][Aa][Tt][Ee][5]
Tpl6              [Tt][Ee][Mm][Pp][Ll][Aa][Tt][Ee][6]
Tpl7              [Tt][Ee][Mm][Pp][Ll][Aa][Tt][Ee][7]
Tpl8              [Tt][Ee][Mm][Pp][Ll][Aa][Tt][Ee][8]
GetAllClass       [Gg][Ee][Tt][ ]+[Aa][Ll][Ll][ ]+[Cc][Ll][Aa][Ss][Ss]
And               [Aa][Nn][Dd]
GenName           [Gg][Ee][Nn][Nn][Aa][Mm][Ee]
FacName           [Ff][Aa][Cc][Nn][Aa][Mm][Ee]
AttName           [Aa][Tt][Tt][Nn][Aa][Mm][Ee]
GenValue          [Gg][Ee][Nn][Vv][Aa][Ll][Uu][Ee]
FacValue          [Ff][Aa][Cc][Vv][Aa][Ll][Uu][Ee]
AttValue          [Aa][Tt][Tt][Vv][Aa][Ll][Uu][Ee]
IsCompound        [Ii][Ss][ ]+[Cc][Oo][Mm][Pp][Oo][Uu][nN][dD]
BelongToCompound  [Bb][Ee][Ll][Oo][Nn][Gg][ ]+[Tt][Oo][ ]+
[Cc][Oo][Mm][Pp][Oo][Uu][nN][dD]
CrlName           [Cc][Rr][Ll][Nn][Aa][Mm][Ee]
RestrictedTo      [Rr][Ee][Ss][Tt][Rr][Ii][Cc][Tt][Ee][Dd][ ]+[Tt][Oo]
Id                [\.\+\/\*\@\;\:\\\(\)\_\-0-9a-zA-Z]+
CardNumber        #[0-9]+

%s LISTOFIDENT    /* specific state to handle lists of identifiers interleaved by
white space */

%%
/* rules section: pattern  { action } */

<LISTOFIDENT>{Id} {strcpy(yylval.STR,yytext);return(IDENT);}

/* tokens marking the unsuccessful or successful end of a query; */
/* they are not kept in the final query string */
[Aa][Bb][Oo][Rr][Tt]     {return(ABORT);}
[Cc][Hh][Ee][Cc][Kk]     {return(CHECK);}

/* building blocks of any Template */
{Tpl1}            {yylval.INT = TEMPLATE1;return(TEMPLATE1);}
{Tpl2}            {yylval.INT = TEMPLATE2;return(TEMPLATE2);}
{Tpl3}            {yylval.INT = TEMPLATE3;return(TEMPLATE3);}
{Tpl4}            {yylval.INT = TEMPLATE4;return(TEMPLATE4);}
{Tpl5}            {yylval.INT = TEMPLATE5;return(TEMPLATE5);}
{Tpl6}            {yylval.INT = TEMPLATE6;return(TEMPLATE6);}
{Tpl7}            {yylval.INT = TEMPLATE7;return(TEMPLATE7);}
{Tpl8}            {yylval.INT = TEMPLATE8;return(TEMPLATE8);}
{GetAllClass}     {return(GETALLCLASS);}
```

```
{And}              {return(AND);}
{GenName}          {yylval.INT = GENNAME;return(GENNAME);}
{FacName}          {yylval.INT = FACNAME;return(FACNAME);}
{AttName}          {yylval.INT = ATTNAME;return(ATTNAME);}
{GenValue}         {yylval.INT = GENVALUE;return(GENVALUE);}
{FacValue}         {yylval.INT = FACVALUE;return(FACVALUE);}
{AttValue}         {yylval.INT = ATTVALUE;return(ATTVALUE);}
{IsCompound}       {return(ISCOMPOUND);}
{BelongToCompound} {return(BELONGTOCOMPOUND);}
{CrlName}          {yylval.INT = CRLNAME;return(CRLNAME);}
[Oo][Rr]           {return(OR);}
{RestrictedTo}     {return(RESTRICTEDTO);}
[Ll][Ii][Nn][Kk][Ee][Dd][ ]+[Bb][Yy] {return(LINKEDBY);}
[Ww][Ii][Tt][Hh]   {return(WITH);}
[Ll][Ii][Nn][Kk][Ee][Dd][ ]+[Tt][Oo] {return(LINKEDTO);}
[Bb][Yy]           {return(BY);}
[Pp][Hh][Aa][Nn][Aa][Mm][Ee] {yylval.INT = PHANAME;return(PHANAME);}
[Ss][Ll][Cc][Nn][Aa][Mm][Ee] {yylval.INT = SLCNAME;return(SLCNAME);}
[Cc][Oo][Nn][Cc][Ee][Pp][Tt][Dd][Ii][Ss][Tt] {return (CONDIST);}
{CardNumber}       {yylval.INT = atoi(yytext+1); return(NUMBER);}
[0-9]+[%]          {yytext[yyleng-1]='\0'; yylval.INT = atoi(yytext); return
(NUMBER);}
{Id}               {strcpy(yylval.STR,yytext);return(IDENT);}
"="                {return(yytext[0]);}
">"                {return(yytext[0]);}
";"                {return(yytext[0]);}
","                {return(yytext[0]);}
"\|"               {return(yytext[0]);}
":"                {return(yytext[0]);}
<LISTOFIDENT>"\"" {BEGIN INITIAL;return(yytext[0]);}
"\""               {BEGIN LISTOFIDENT;return(yytext[0]);}
%%

/* no main; Yacc generated code will call yylex */
```

As it will be seen, the Lexical Analyzer which can be generated from this file does not work alone, but cooperatively with a Yacc generated parser.

2.2 The iqs.y file (using Yacc)

The unified grammar for both the Assisted Mode and Batch Mode IQLs is obtained by moving up the root of both grammars to a common level. This is feasible because the set of valid tokens for both grammars is exactly the same. Only the possible valid sequences are different.

Every function implementing a semantic action has the prefix `iqsSA`, that is, IQS Semantic Action[10].

The contents of file **iqs.y**, with a description of the unified IQL grammar acceptable by Yacc, follow:

[10]to know more about the Semantic Actions refer to chapter **6 The IQS Semantic Actions API (actions.c)**.

```
%{
/* definition section */
#include "actions.h"   /* semantic actions module header */

%}

/* possible types for yylval (the token recognized by Lex) */
%union {
    int INT;
    char STR[255];
}

/* type definition for each grammar token */
%token <INT> TEMPLATE1 TEMPLATE2 TEMPLATE3 TEMPLATE4 TEMPLATE5 TEMPLATE6
             TEMPLATE7 TEMPLATE8

%token <INT> GENNAME ATTNAME FACNAME GENVALUE ATTVALUE FACVALUE SLCNAME PHANAME
             CRLNAME NUMBER

%token <STR> IDENT

%token ABORT CHECK
%token GETALLCLASS AND CONDIST ISCOMPOUND BELONGTOCOMPOUND OR RESTRICTEDTO
%token LINKEDBY WITH LINKEDTO BY

/* type for the non-terminal ListOfIdent */
%type <STR> ListOfIDENT

%%

/* rules section */

/* ROOT of the unified grammar */
Iqs    :    batchIqs
       |    assistIqs
       ;

/* BATCH mode branch */

batchIqs :    batchIqs batchTemplate1 { iqsSAcheck(); }
         |    batchIqs batchTemplate2 { iqsSAcheck(); }
         |    batchIqs batchTemplate3 { iqsSAcheck(); }
         |    batchIqs batchTemplate4 { iqsSAcheck(); }
         |    batchIqs batchTemplate5 { iqsSAcheck(); }
         |    batchIqs batchTemplate6 { iqsSAcheck(); }
         |    batchIqs batchTemplate7 { iqsSAcheck(); }
         |    batchIqs batchTemplate8 { iqsSAcheck(); }
         |    batchTemplate1 { iqsSAcheck(); }
         |    batchTemplate2 { iqsSAcheck(); }
         |    batchTemplate3 { iqsSAcheck(); }
         |    batchTemplate4 { iqsSAcheck(); }
         |    batchTemplate5 { iqsSAcheck(); }
         |    batchTemplate6 { iqsSAcheck(); }
         |    batchTemplate7 { iqsSAcheck(); }
         |    batchTemplate8 { iqsSAcheck(); }
         ;
```

```
/* ASSIST mode branch */

assistIqs:    assistTemplate1
         |    assistTemplate2
         |    assistTemplate3
         |    assistTemplate4
         |    assistTemplate5
         |    assistTemplate6
         |    assistTemplate7
         |    assistTemplate8
         |    CHECK { iqsSAcheck(); }
         |    ABORT { iqsSAabort(); }
         ;

/* COMMON rules */

ListOfIDENT :   IDENT {strcpy($$,$1);}
            |   ListOfIDENT IDENT {strcat($1," ");strcat($1,$2);strcpy($$,$1);}
            ;

/*-----------------------Template1 BATCH mode rule----------------------*/

batchTemplate1    :    NUMBER { batchIqsSAcheckIndex($1); } TEMPLATE1
                       { iqsSAinitGetAllClass($3); } GETALLCLASS '=' '\"'
                       ListOfIDENT '\"' { iqsSAgetAoidsBellowClass($8); }
                       batchTemplate11
                  ;

batchTemplate11   :
                  |    AND batchAttrDesc
                  ;

batchAttrDesc   :    batchGenDesc batchTemplate11
                |    batchFacDesc batchTemplate11
                |    batchAttDesc batchTemplate11
                ;

batchGenDesc    :    GENNAME { iqsSAafterAttrTypeChoice($1); } '=' '\"'
                     ListOfIDENT '\"' { iqsSAgetAttrValues($1,$5); } AND
                     GENVALUE '=' '\"' ListOfIDENT '\"'
                     { iqsSAgetAoidsByValue($9, $5, $12, -1); }
                ;

batchFacDesc    :    FACNAME { iqsSAafterAttrTypeChoice($1); } '=' '\"'
                     ListOfIDENT '\"' { iqsSAgetAttrValues($1,$5); } AND
                     FACVALUE '=' '\"' ListOfIDENT '\"'
                     AND CONDIST '>''=' NUMBER
                     { iqsSAgetAoidsByValue($9, $5, $12, $18); }
                ;

batchAttDesc    :    ATTNAME { iqsSAafterAttrTypeChoice($1); } '=' '\"'
                     ListOfIDENT '\"' { iqsSAgetAttrValues($1,$5); } AND
                     ATTVALUE '=' '\"' ListOfIDENT '\"'
                     { iqsSAgetAoidsByValue($9, $5, $12, -1); }
                ;
```

```
/*---------------------Template1 ASSISTED mode rule--------------------*/

assistTemplate1   :    TEMPLATE1 { iqsSAinitGetAllClass($1); }
                  |    TEMPLATE1 GETALLCLASS '=' '\"' ListOfIDENT '\"'
                       { iqsSAgetAoidsBellowClass($5); }
                  |    TEMPLATE1 GETALLCLASS '=' '\"' ListOfIDENT '\"'
                       AND AttrDesc
                  ;

AttrDesc    :    AttrName
            |    AttrNameAnd AttrDesc
            |    AttrNameEqId
            |    AttrNameEqIdAnd AttrDesc
            |    AttrNameEqIdAndAttrValueEqId
            |    AttrNameEqIdAndAttrValueEqIdAnd AttrDesc
            ;

AttrName    :    GENNAME { iqsSAafterAttrTypeChoice($1); }
            |    FACNAME { iqsSAafterAttrTypeChoice($1); }
            |    ATTNAME { iqsSAafterAttrTypeChoice($1); }
            ;

AttrNameAnd :    GENNAME AND
            |    FACNAME AND
            |    ATTNAME AND
            ;

AttrNameEqId:    GENNAME '=' '\"' ListOfIDENT '\"' {iqsSAgetAttrValues($1,$4); }
            |    FACNAME '=' '\"' ListOfIDENT '\"' {iqsSAgetAttrValues($1,$4); }
            |    ATTNAME '=' '\"' ListOfIDENT '\"' {iqsSAgetAttrValues($1,$4); }
            ;

AttrNameEqIdAnd:    GENNAME '=' '\"' ListOfIDENT '\"' AND
               |    FACNAME '=' '\"' ListOfIDENT '\"' AND
               |    ATTNAME '=' '\"' ListOfIDENT '\"' AND
               ;

AttrNameEqIdAndAttrValueEqId:    GENNAME '=' '\"' ListOfIDENT '\"' AND GENVALUE
                                 '=' '\"' ListOfIDENT '\"'
                                 { iqsSAgetAoidsByValue($7, $4, $10, -1); }
                            |    FACNAME '=' '\"' ListOfIDENT '\"' AND FACVALUE
                                 '=' '\"' ListOfIDENT '\"'
                                 AND CONDIST '>''=' NUMBER
                                 { iqsSAgetAoidsByValue($7, $4, $10, -1);}
                            |    ATTNAME '=' '\"' ListOfIDENT '\"' AND ATTVALUE
                                 '=' '\"' ListOfIDENT '\"'
                                 { iqsSAgetAoidsByValue($7, $4, $10, -1); }
                            ;

AttrNameEqIdAndAttrValueEqIdAnd:    GENNAME '=' '\"' ListOfIDENT '\"' AND
                                    GENVALUE '=' '\"' ListOfIDENT '\"' AND
                               |    FACNAME '=' '\"' ListOfIDENT '\"' AND
                                    FACVALUE '=' '\"' ListOfIDENT '\"' AND
                                    CONDIST '>''=' NUMBER AND
                               |    ATTNAME '=' '\"' ListOfIDENT '\"' AND
                                    ATTVALUE '=' '\"' ListOfIDENT '\"' AND
                               ;
```

```
/*-----------------------Template2 BATCH mode rule----------------------*/

batchTemplate2    :    NUMBER { batchIqsSAcheckIndex($1); } TEMPLATE2
                       { iqsSAinitGetAllClass($3); } GETALLCLASS '=' '\"'
                       ListOfIDENT '\"' { iqsSAgetAoidsBellowClass($8); }
                       batchTemplate22
                  ;

batchTemplate22   :
                  |    AND batcht2AttrDesc
                  ;

batcht2AttrDesc   :    batchGenDesc batchTemplate22
                  |    batchFacDesc batchTemplate22
                  |    batchAttDesc batchTemplate22
                  |    batchSlcDesc batchTemplate11
                  |    batchPhaDesc batchTemplate22
                  ;

batchSlcDesc      :    SLCNAME { iqsSAafterAttrTypeChoice($1); } '=' '\"'
                       ListOfIDENT '\"' { iqsSAgetAoidsBySlc($5); }
                  ;

batchPhaDesc      :    PHANAME { iqsSAafterAttrTypeChoice($1); } '=' '\"'
                       ListOfIDENT '\"' { iqsSAgetSlcsAndAoidsByPha($5); }
                  ;

/*-----------------------Template2 ASSISTED mode rule------------------*/

assistTemplate2   :    TEMPLATE2 { iqsSAinitGetAllClass($1); }
                  |    TEMPLATE2 GETALLCLASS '=' '\"' ListOfIDENT '\"'
                       { iqsSAgetAoidsBellowClass($5); }
                  |    TEMPLATE2 GETALLCLASS '=' '\"' ListOfIDENT '\"' AND
                       t2AttrDesc
                  ;

t2AttrDesc  :    AttrName
            |    AttrNameAnd t2AttrDesc
            |    AttrNameEqId
            |    AttrNameEqIdAnd t2AttrDesc
            |    AttrNameEqIdAndAttrValueEqId
            |    AttrNameEqIdAndAttrValueEqIdAnd t2AttrDesc
            |    SlcDesc
            |    PhaDesc
            ;

SlcDesc     :    SLCNAME { iqsSAafterAttrTypeChoice($1); }
            |    SLCNAME AND t2AttrDesc
            |    SLCNAME '=' '\"' ListOfIDENT '\"' { iqsSAgetAoidsBySlc($4); }
            |    SLCNAME '=' '\"' ListOfIDENT '\"' AND t2AttrDesc
            ;

PhaDesc     :    PHANAME { iqsSAafterAttrTypeChoice($1); }
            |    PHANAME AND t2AttrDesc
            |    PHANAME '=' '\"' ListOfIDENT '\"'
                 { iqsSAgetSlcsAndAoidsByPha($4); }
            |    PHANAME '=' '\"' ListOfIDENT '\"' AND t2AttrDesc
            ;
```

```
/*----------------------Template3 BATCH mode rule---------------------*/

batchTemplate3    :    NUMBER { batchIqsSAcheckIndex($1); } TEMPLATE3
                       { iqsSAinitGetAllClass($3); } GETALLCLASS '=' '\"'
                       ListOfIDENT '\"' { iqsSAgetAoidsBellowClass($8); }
                       batchTemplate33
                  ;

batchTemplate33   :
                  |    AND batchAttrDesc batchTemplate333
                  |    ANDISCOMPOUND { iqsSAafterIsCompoundPressed();}
                       batchTemplate3333
                  ;

batchTemplate333  :
                  |    ANDISCOMPOUND { iqsSAafterIsCompoundPressed();}
                       batchTemplate3333
                  ;

batchTemplate3333 :
                  |    AND batchCAttrDesc
                  ;

batchCAttrDesc    :    batchGenDesc batchTemplate3333
                  |    batchFacDesc batchTemplate3333
                  |    batchAttDesc batchTemplate3333
                  |    batchCrlDesc batchTemplate3333

batchCrlDesc    :    CRLNAME { iqsSAafterAttrTypeChoice($1); } '=' '\"'
                     ListOfIDENT '\"' { iqsSAgetAoidsByCaractRel($5); }
                ;

/*----------------------Template3 ASSISTED mode rule-----------------*/

assistTemplate3   :    TEMPLATE3 { iqsSAinitGetAllClass($1); }
                  |    TEMPLATE3 GETALLCLASS '=' '\"' ListOfIDENT '\"'
                       { iqsSAgetAoidsBellowClass($5); }
                  |    TEMPLATE3 GETALLCLASS '=' '\"' ListOfIDENT '\"' AND
                       AttrDesc
                  |    TEMPLATE3 GETALLCLASS '=' '\"' ListOfIDENT '\"' AND
                       AttrDescBeforeC ISCOMPOUND
                       { iqsSAafterIsCompoundPressed(); }
                  |    TEMPLATE3 GETALLCLASS '=' '\"' ListOfIDENT '\"' AND
                       AttrDescBeforeC ISCOMPOUND AND CAttrDesc
                  |    TEMPLATE3 GETALLCLASS '=' '\"' ListOfIDENT '\"' AND
                       ISCOMPOUND { iqsSAafterIsCompoundPressed(); }
                  |    TEMPLATE3 GETALLCLASS '=' '\"' ListOfIDENT '\"' AND
                       ISCOMPOUND AND CAttrDesc
                  ;

AttrDescBeforeC   : AttrNameAnd
                  | AttrNameAnd AttrDescBeforeC
                  | AttrNameEqIdAnd
                  | AttrNameEqIdAnd AttrDescBeforeC
                  | AttrNameEqIdAndAttrValueEqIdAnd
                  | AttrNameEqIdAndAttrValueEqIdAnd AttrDescBeforeC
                  ;
```

```
CAttrDesc   :    AttrName
            |    AttrNameAnd CAttrDesc
            |    AttrNameEqId
            |    AttrNameEqIdAnd CAttrDesc
            |    AttrNameEqIdAndAttrValueEqId
            |    AttrNameEqIdAndAttrValueEqIdAnd CAttrDesc
            |    CRLNAME { iqsSAafterAttrTypeChoice($1); }
            |    CRLNAME AND CAttrDesc
            |    CRLNAME '=' '\"' ListOfIDENT '\"'
                 { iqsSAgetAoidsByCaractRel($4); }
            |    CRLNAME '=' '\"' ListOfIDENT '\"' AND CAttrDesc
            ;

/*-----------------------Template4 BATCH mode rule----------------------*/

batchTemplate4    :    NUMBER { batchIqsSAcheckIndex($1); } TEMPLATE4
                       { iqsSAinitGetAllClass($3); } GETALLCLASS '=' '\"'
                       ListOfIDENT '\"' { iqsSAgetAoidsBellowClass($8); }
                       batchTemplate44
                  ;

batchTemplate44   :
                  |    AND batchAttrDesc batchTemplate444
                  |    ANDBELONGTOCOMPOUND
                       { iqsSAafterBelongToCompoundPressed(); }
                       batchTemplate3333
                  ;

batchTemplate444  :
                  |    ANDBELONGTOCOMPOUND
                       { iqsSAafterBelongToCompoundPressed(); }
                       batchTemplate3333
                   ;

/*-----------------------Template4 ASSISTED mode rule------------------*/

assistTemplate4   :    TEMPLATE4 { iqsSAinitGetAllClass($1); }
                  |    TEMPLATE4 GETALLCLASS '=' '\"' ListOfIDENT '\"'
                       { iqsSAgetAoidsBellowClass($5); }
                  |    TEMPLATE4 GETALLCLASS '=' '\"' ListOfIDENT '\"' AND
                       AttrDesc
                  |    TEMPLATE4 GETALLCLASS '=' '\"' ListOfIDENT '\"' AND
                       AttrDesc BeforeC BELONGTOCOMPOUND
                       { iqsSAafterBelongToCompoundPressed(); }
                  |    TEMPLATE4 GETALLCLASS '=' '\"' ListOfIDENT '\"' AND
                       AttrDescBeforeC BELONGTOCOMPOUND AND CAttrDesc
                  |    TEMPLATE4 GETALLCLASS '=' '\"' ListOfIDENT '\"'
                       BELONGTOCOMPOUND
                       { iqsSAafterBelongToCompoundPressed(); }
                  |    TEMPLATE4 GETALLCLASS '=' '\"' ListOfIDENT '\"'
                       BELONGTOCOMPOUND AND CAttrDesc
                  ;
```

```
/*-----------------------Template5 BATCH mode rule----------------------*/

batchTemplate5    :    NUMBER { batchIqsSAcheckIndex($1); } TEMPLATE5
                       { iqsSAinitQuery($3); } NUMBER
                       { iqsSAgetAoidsFromQuery($5); } batchTemplate55
                  ;

batchTemplate55   :
                  |    LINKEDBY '\"' ListOfIDENT '\"'
                       { iqsSAgetSourcesByLink($3); } batchTemplate555
                  ;

batchTemplate555  :
                  |    WITH NUMBER { #ifdef __DOS__
                   iqsSAgetSourcesByLinkAndSinks(iqsState.iqsLNKNames.names,$2);
                                    #endif }
                  ;

/*-----------------------Template5 ASSISTED mode rule------------------*/

assistTemplate5   :    TEMPLATE5 { iqsSAinitQuery($1); }
                  |    TEMPLATE5 NUMBER { iqsSAgetAoidsFromQuery($2); }
                  |    TEMPLATE5 NUMBER LINKEDBY '\"' ListOfIDENT '\"'
                       { iqsSAgetSourcesByLink($5); }
                  |    TEMPLATE5 NUMBER LINKEDBY '\"' ListOfIDENT '\"' WITH
                       NUMBER { iqsSAgetSourcesByLinkAndSinks($5,$8); }
                  ;

/*-----------------------Template6 BATCH mode rule----------------------*/

batchTemplate6    :    NUMBER { batchIqsSAcheckIndex($1); } TEMPLATE6
                       { iqsSAinitQuery($3); } NUMBER
                       { iqsSAgetAoidsFromQuery($5); } batchTemplate66
                  ;

batchTemplate66   :
                  |    LINKEDTO NUMBER { iqsSAgetSourcesAndLinksBySinks($2); }
                       batchTemplate666
                  ;

batchTemplate666  :
                  |    BY '\"' ListOfIDENT '\"' { iqsSAgetSourcesByLink($3); }
                  ;

/*-----------------------Template6 ASSISTED mode rule------------------*/

assistTemplate6   :    TEMPLATE6 { iqsSAinitQuery($1); }
                  |    TEMPLATE6 NUMBER { iqsSAgetAoidsFromQuery($2); }
                  |    TEMPLATE6 NUMBER LINKEDTO NUMBER
                       { iqsSAgetSourcesAndLinksBySinks($4); }
                  |    TEMPLATE6 NUMBER LINKEDTO NUMBER BY '\"' ListOfIDENT '\"'
                       { iqsSAgetSourcesByLink($7); }
                  ;
```

```
/*----------------------Template7 BATCH mode rule---------------------*/

batchTemplate7    :    NUMBER { batchIqsSAcheckIndex($1); } TEMPLATE7
                       { iqsSAinitQuery($3); } NUMBER
                       { iqsSAgetAoidsFromQuery($5); } batchTemplate77
                  ;

batchTemplate77   :
                  |    OR NUMBER { iqsSAqueryUnion($2); }
                  ;

/*----------------------Template7 ASSISTED mode rule-----------------*/

assistTemplate7   :    TEMPLATE7 { iqsSAinitQuery($1); }
                  |    TEMPLATE7 NUMBER { iqsSAgetAoidsFromQuery($2); }
                  |    TEMPLATE7 NUMBER OR NUMBER { iqsSAqueryUnion($4); }
                  ;

/*----------------------Template8 BATCH mode rule---------------------*/

batchTemplate8    :   NUMBER { batchIqsSAcheckIndex($1); }
                      TEMPLATE8 { iqsSAinitQuery($3); } NUMBER
                      { iqsSAgetAoidsFromQuery($5); } batchTemplate88
                  ;

batchTemplate88   :
                  |   RESTRICTEDTO batchAttrDesc
                  ;

/*----------------------Template8 ASSISTED mode rule-----------------*/

assistTemplate8   :    TEMPLATE8 { iqsSAinitQuery($1); }
                  |    TEMPLATE8 NUMBER { iqsSAgetAoidsFromQuery($2); }
                  |    TEMPLATE8 NUMBER RESTRICTEDTO AttrDesc
                  ;

%%

#include "myfunc.c" /* mygetc, myputc, yyerror and yywrap (re)definitions; */

/* joining the Lex generated code at the end of the Yacc generated code */
/* and producing a unique module containing the scanner and the parser  */
#ifdef __DOS__
#include "lexyy.c"
#else
#include "lex.yy.c"
#endif
```

2.3 Lexer - Parser communication

When a Lex scanner and a Yacc parser are used in a cooperative way, the Lex scanner main routine, **yylex**, acts as a subroutine of **yyparse**, the main routine of the Yacc parser.

The Lexer scans the input string for the character patterns specified in his rule's section. Whenever a valid pattern (or token), is found it returns a token specific integer code to Yacc, and optionally the token itself via one of the fields (token data type dependent) of the **yylval** union, defined in **iqs.y**. Then, Yacc manages to match the token code within one of his grammar rules. If the matching is successful, the eventual semantic action is executed.

The token specific integer codes (which must be known by both the Lexer and the Parser) are only generated for those tokens specified in the `%token` declarations in the `/* definition section */` of **iqs.y**.

The file **ytab.h**, containing that (and eventually other) information that must be shared by the Lexer and the Parser, is shown next:

```
#ifdef __DOS__
# define TEMPLATE1 257
# define TEMPLATE2 258
# define TEMPLATE3 259
# define TEMPLATE4 260
# define TEMPLATE5 261
# define TEMPLATE6 262
# define TEMPLATE7 263
# define TEMPLATE8 264
# define GENNAME 265
# define ATTNAME 266
# define FACNAME 267
# define GENVALUE 268
# define ATTVALUE 269
# define FACVALUE 270
# define SLCNAME 271
# define PHANAME 272
# define CRLNAME 273
# define NUMBER 274
# define IDENT 275
# define ABORT 276
# define CHECK 277
# define GETALLCLASS 278
# define AND 279
# define CONDIST 280
# define ISCOMPOUND 281
# define BELONGTOCOMPOUND 282
# define OR 283
# define RESTRICTEDTO 284
# define LINKEDBY 285
# define WITH 286
# define LINKEDTO 287
# define BY 288
#endif
```

2.4 Structure of the IQL grammar

The Assisted Mode branch and the Batch Mode branch of the unified IQL grammar, as depicted in the **iqs.y** file, despite sharing the tokens (and thus based on the same Lexer), deeply differ in their structure. They were tailored to fulfill two different (although complementary) modes of operation: batch or interactive query resolution.

Both grammars are left-recursive in order to let the parsing process to be a little more efficient.

2.4.1 Assisted Mode IQL sub-grammar issues

Being interface-event dependent, the Assisted Mode sub-grammar has to deal with a few specific issues:

• **redundancy**: a sequence of repeated tokens (for instance originated by repeated mouse clicks in the same button) should produce the same result as one instance of the same token. Redundancy is a very often situation in the Interface layer, but it is not handled there; instead, it must be recognized by the Parser layer which, via semantic actions, will have to prevent it from remaining in the query text and so providing for compatibility with Batch Mode;

• **incompleteness**: a sequence of one or more tokens may not imply any object filtering; instead, they could stand for a valid sequence of interface actions not producing any refinement of the present query solution. Again, and for compatibility with Batch Mode, incompleteness is not allowed to stay in the final query text. Only those tokens whose recognition implied object filtering will remain.

• **alternation between Interface Layer and Parser Layer** control over the IQS module. This feature reflects an implementation constraint: it would be desirable to have the Lexer in background, in an endless loop, waiting for tokens to be recognized, while the Interface Layer would provide for feeding it. In fact, Lex & Yacc were tailored for this kind of behaviour, but, in the cooperative multitasking environment of WINDOWS 3.1, which lacks the notion of process (or preemptive multitasking), this introduces communication and synchronization problems between the two Layers in the case we want them to work concurrently[11]. An easier way is to let the control alternate between the Interface Layer and the Parser Layer. Every time an Interface event produces a token (or a small set of tokens, not semantically dividable), the sentence so far built is added that token and again submitted to the Parser Layer. The IQL Assisted Mode sub-grammar will be highly partitioned because it must predict every valid situation in which the query phrase can grow. Also,

[11]Threads could have been a valid solution to this problem; however, in WINDOWS 3.1, threads are hard to code and to maintain.

the semantic actions will be terminal, that is, they will refer only to the token (or small set of tokens) last added to the query text. Therefore, the rules of the IQL Assisted Mode sub-grammar assume a **stair-fashion**, reflecting this behaviour.

A piece of **iqs.y**, with an Assisted Mode rule, will help to make these details clear:

```
/*--------------------Template1 ASSISTED mode rule-------------------*/

/* Note the terminal semantic actions ... */
assistTemplate1   :    TEMPLATE1 { iqsSAinitGetAllClass($1); }
                  |    TEMPLATE1 GETALLCLASS '=' '\"' ListOfIDENT '\"'
                       { iqsSAgetAoidsBellowClass($5); }
                  |    TEMPLATE1 GETALLCLASS '=' '\"' ListOfIDENT '\"'
                       AND AttrDesc
                  ;

/* Note the stair fashion; every valid growing possibilities for a sentence  */
/* which at least matched TEMPLATE1 GETALLCLASS '=' '\"' ListOfIDENT '\"' AND*/
/* are considered */

AttrDesc    :    AttrName
            |    AttrNameAnd AttrDesc
            |    AttrNameEqId
            |    AttrNameEqIdAnd AttrDesc
            |    AttrNameEqIdAndAttrValueEqId
            |    AttrNameEqIdAndAttrValueEqIdAnd AttrDesc
            ;

/* To exemplify redundancy and incompleteness take a look at the next two    */
/* rules: */

AttrName    :    GENNAME { iqsSAafterAttrTypeChoice($1); }
            |    FACNAME { iqsSAafterAttrTypeChoice($1); }
            |    ATTNAME { iqsSAafterAttrTypeChoice($1); }
            ;

AttrNameAnd :    GENNAME AND
            |    FACNAME AND
            |    ATTNAME AND
            ;

/* It is then possible to have TEMPLATE1 GETALLCLASS '=' '\"' ListOfIDENT '\"'*/
/* AND GENNAME AND GENAME ... etc; this introduces:                          */
/* redundancy: the effect is the same of having just one GENNAME token;       */
/*             the semantic action { iqsSAafterAttrTypeChoice($1); } is       */
/*             executed twice;                                                */
/* incompleteness: the semantic action { iqsSAafterAttrTypeChoice($1); }      */
/*                 executed every time GENNAME is recognized, does not refine */
/*                 the present query solution;                                */
/* The Minimal Efficient Form[12], resulting from eliminating redundancy and  */
/* incompleteness would be TEMPLATE1 GETALLCLASS '=' '\"' ListOfIDENT '\"'   */
```

[12] refer to **5 Interface Query Language** at [IQS-2.1].

2.4.2 Batch Mode IQL sub-grammar issues

This branch of the unified IQL grammar can be directly derived from the Templates specification. This sub-grammar is simpler and smaller than the Assisted Mode one, because:

- since it handles only Minimal Efficient Forms, it does not have to deal with redundancy and incompleteness;

- the Batch Mode is a kind of "silent" operation mode: the Interface Layer gives the control of the execution flow to the Parser Layer every time a batch of queries has to be recognized and solved; then, it waits patiently until all the queries in the batch are solved or an error occurs. There's no alternation between the Interface Layer and the Parser Layer, therefore semantic actions can alternate with tokens within the grammar rules.

For instance, consider the three rules for the non-terminal `assistTemplate1` in the previous example with `Template1 ASSISTED mode rule` against the equivalent rule in Batch Mode:

```
/*----------------------Template1 BATCH mode rule---------------------*/

batchTemplate1    :    NUMBER { batchIqsSAcheckIndex($1); } TEMPLATE1
                       { iqsSAinitGetAllClass($3); } GETALLCLASS '=' '\"'
                       ListOfIDENT '\"' { iqsSAgetAoidsBellowClass($8); }
                       batchTemplate11
                  ;

/* etc ... */
```

3 Importing the generated code to a Windows DLL

The automatically generated code for the Lexer and the Parser is not ready to be directly used in a WINDOWS 3.1 DLL. This chapter describes the necessary modifications.

3.1 Redirecting the Lexer input

This section is based on the work described at [GF93].

By default, the lexical analyzer `input()` macro scans through the standard input using the call `getc(yyin)`. In the **lexyy.c** file, both the `input()` macro and the `yyin` declarations are:

```
# define input() (((yytchar=yysptr>yysbuf?U(*--y ysptr):getc(yyin)==10?
(yylineno++,yytchar):yytchar)==EOF?0:yytchar)

FILE *yyin = {stdin};
```

Instead of scanning *stdin*, what we really want is the Lexer to check for the tokens in a memory string; this string will be the query synthesized in Assisted Mode or one of a batch of queries in the Batch Mode.

In order to redirect the Lexer input from the *stdin* to a memory string, the automatically generated **yylex.c** file must suffer a few modifications:

• substitute the `FILE *yyin = {stdin};` declaration by `char *yyin;`

• in the macro definition of `input()` substitute `getc(yyin)` by `mygetc(yyin++)` and `EOF` by `0`.

That is, the `input()` macro and the `yyin` declaration should be:

```
# define input() (((yytchar=yysptr>yysbuf?U(*--yysptr):mygetc(yyin++))==10?
(yylineno++,yytchar):yytchar)==0?0:yytchar)

char *yyin;
```

The `mygetc` function is:

```
int mygetc(char * strin)
{
 int c;
 return(c=*strin);
}
```

It is defined in the **myfunc.c** file, which has some other useful redefinitions[13].

All that is necessary now is to make `yyin` point to our string in memory before `yylex()` starts the Lexer. Once the `yyparse()` function calls `yylex()`, that can be done in the Parser file generated by Yacc: **ytab.c**. All we need is:

• to modify the `yyparse` function
  from `int yyparse(void)`
  to `int yyparse(char *str)`
in order to let `yyparse` receive the string to be scanned as a parameter;

• to add `yyin=str;` to the body of the `yyparse` function, but before `yylex` is invoked.

[13]refer to **3.3 The myfunc.c redefinitions file**

Figure 3.1 and 3.2 show `yyparse` as generated by Yacc and after these modifications, respectively.

```
/* yyparse AS GENERATED BY YACC */

/*
** yyparse - return 0 if worked, 1 if syntax error not recovered from
*/
int
yyparse()
{
    register YYSTYPE *yypvt;    /* top of value stack for $vars */
    unsigned yymaxdepth = YYMAXDEPTH;

    /*
    ** Initialize externals - yyparse may be called more than once
    */
    yyv = (YYSTYPE*)malloc(yymaxdepth*sizeof(YYSTYPE));
    yys = (int*)malloc(yymaxdepth*sizeof(int));

            /* REMAINING yyparse CODE */
}
```

Figure 3.1 - `yyparse` as generated by Yacc

```
/* yyparse AFTER MODIFICATIONS */

/*
** yyparse - return 0 if worked, 1 if syntax error not recovered from
*/
int
yyparse(char *str)  /* HEADER REDEFINITION */
{
    register YYSTYPE *yypvt;    /* top of value stack for $vars */
    unsigned yymaxdepth = YYMAXDEPTH;

    /*
    ** Initialize externals - yyparse may be called more than once
    */
    yyv = (YYSTYPE*)malloc(yymaxdepth*sizeof(YYSTYPE));
    yys = (int*)malloc(yymaxdepth*sizeof(int));
    yyin = str;  /* yyin REDIRECTION */

            /* REMAINING yyparse CODE */
}
```

Figure 3.2 - `yyparse` after modifications

3.2 Output in the Lexer and in the Parser

Output in the **lexyy.c** and in the **ytab.c** generated files must be also carefully controlled. By default, *stdout* is used. Once that is unacceptable in a WINDOWS 3.1 DLL, we must avoid *stdout* based output.

In the Parser generated file, **ytab.c**, all the code that writes to the *stdout* is isolated between the `#if YYDEBUG` and `#endif` pre-processor directives. Therefore, compiling the file without defining `YYDEBUG` solves the problem (there's an exception concerning the function `yyerror`; see section 3.3 for more details).

In the Lexer generated file, **lexyy.c**, the `#if LEXDEBUG` and `#endif` pre-processor directives define almost every code that writes in the *stdout*, and like in **ytab.c**, not defining `LEXDEBUG` during compilation prevents access to *stdout*. However, there are also three other situations that must be handled:

1. the `output` macro must be redefined:

from `# define output(c) putc(c,yyout)`

to `# define output(c) myputc(c)`

because `yyout` is declared as `FILE *yyout = {stdout};`
The `myputc` function is:

```
int myputc()
{
 return(1);
}
```

and like `mygetc` is defined in the **myfun.c** file[14]

2. the `ECHO` macro, defined as `# define ECHO fprintf(yyout, "%s",yytext)`, should not be used anywhere in the **lexyy.c** file (it's enough not to use it in the action C code for a pattern in the **iqs.l** file);

3. the last four lines of the function `yylex` are:

```
default:
fprintf(yyout,"bad switch yylook %d",nstr);
} return(0); }
/* end of yylex */
```

and so the call to `fprintf` must be avoided by nesting that line in a comment, for instance.

[14]refer to **3.3 The myfunc.c redefinitions file**

3.3 The myfunc.c redefinition file

The file **myfunc.c** contains the definition of the `mygetc` and `myputc` functions as well as the redefinition of the `yyerror` function used by **ytab.c** and the redefinition of the `yywrap` function used by **lexyy.c**.

By default, the `yyerror` function, called in some error situations during lexical analysis, prints a message in the *stdout*. Because we must avoid that in the DLL environment, we redefined the function using a pre-processor directive to code the environment in which it is being executed.

The `yywrap` function tells what to do when encountering the end of the string (or the end of file when `yyin` is a `FILE*` pointing to *stdin*) being scanned. Returning **1** makes the scanner returns a zero token to report the end of the string. This is the default behaviour but this redefinition makes sure that happens.

The contents of the file **myfunc.c** follow:

```
int mygetc(char * strin)
{
 int c;
 return(c=*strin);
}


int myputc()
{
 return(1);
}


int yyerror(char *s)
{
#ifdef __DOS__
  return 1;
#else
    (void)printf("Error: %s\n",s);
#endif
}


int yywrap()
{
    return 1;
}
```

3.4 Importing malloc and realloc

The first line of code of the file **ytab.c** is:

```
extern char *malloc(), *realloc();
```

and it must be nested inside a comment if we want to successfully compile **ytab.c** because

```
#include "actions.h"
```

already includes the libraries for those functions and being that include directive in the definition section of **iqs.y**, is copied verbatim to the beginning of **ytab.c**.

3.5 The #define __DOS__ directive

The automatic code generation using Lex & Yacc took place in a UNIX environment (although DOS versions of these tools can also be found).

Developing the Parser Layer in UNIX allowed for a quick and easy automation of all the previously discussed modifications one needed to perform in order to use the generated code in a DLL: a simple script using common UNIX tools (as **head** and **sed)** was enough. It even was possible to test the parser, checking for grammar construction errors as well as scanning problems.

Although the semantic actions were not implemented at this phase, their prototypes were more or less stable since the design phase and so we could have calls to semantic actions executing nothing. After all, we just wanted to simulate the behaviour of the parser. Their prototypes, however, already reflected some details of the DLL C code style, namely the use of `FAR` pointers as parameters.

Because `FAR` pointers were unknown in the UNIX world (or any other environment not defining that variant), we ended with two prototype definitions for each semantic action: one with `FAR` type pointers (to be compiled when building the DLL in the DOS world) and other with "simple" pointers (UNIX compatible).

The `#define __DOS__` and `#endif` directives allowed for this "double identity" for the semantic actions. Later, when the body of the semantic actions was fully implemented, we could still test just the parsing mechanism because this method could also be used to deactivate the code we didn't want to be compiled (and therefore executed).

In fact, everything we wanted not to be accessible on the UNIX side or in the DOS side could be under this kind of control: to compile the code nested between the `#define __DOS__` and `#endif`, it would be enough to make the `__DOS__` symbol a C preprocessor parameter.

The files **actions.h** and **actions.c**, the header file and the implementation file of the semantic actions, respectively, make full use of the `#define __DOS__` directive. For instance, consider the Figure 4.1 and Figure 4.2, with extracts from **actions.h** and **actions.c**.

```
#ifdef __DOS__
void iqsSAgetAoidsBellowClass(char FAR *);
#else
void iqsSAgetAoidsBellowClass(char *);
#endif
```

Figure 4.1 - a "double identity" semantic action prototype

```
#ifdef __DOS__
void iqsSAgetAoidsBellowClass(char FAR * class)
#else
void iqsSAgetAoidsBellowClass(char * class)
#endif
{

#ifdef __DOS__
    if (iqsState.iqsBatchOn && batchIqsSemanticError()) return;

    /* REMAINING OF iqsSAgetAoidsBellowClass */

#endif

}
```

Figure 4.2 - a "double identity" semantic action body

3.6 The Large model of compilation

The modifications performed over the code generated from Lex & Yacc still aren't enough to make sure the code will execute in a DLL environment. The code will compile, but probably will hang the system with a *General Fault Protection Error* due to bad memory access. DLLs have strict rules concerning memory references and operations, namely:

- external, global or static variables reside in the DLL Data Segment; by default, if not explicitly `FAR`, a memory reference in a DLL is considered to be made relatively to the beginning of his Data Segment, and so taking the address of those type of variables and using it is peaceful;

- function parameters and local variables use the Stack Segment of the caller of the DLL (the DLL has no Stack Segment).

As we saw, the code automatically generated by Lex & Yacc (UNIX or DOS versions) does not use `FAR` pointers. Therefore, if somewhere in that code the address of a function parameter or a local variable is taken, the Stack Segment reference will be lost, only remaining the Offset part, which will be assumed to be relative to the beginning of the DLL Data Segment! This will almost certainly bring problems.

Very often, the compiler will detect these situations and so one could think of modifying *in situ* the code generated by Lex & Yacc. Generated code, however, should not be modified unless there is no alternative.

A simple and effective way to solve the problem is to compile with the Large model. Therefore, all pointers are `FAR` by default, and we don't need to care about the possibility of having the generated code to violate the DLL strict memory access philosophy.

Using the Large memory model does not mean that the code implementing the Semantic actions and the IQS API no longer should follow the DLL rules. In fact, if we compromise with those rules while coding is taking place we can think of using another Parser in the future, perhaps already respecting those rules; therefore, we would have the other modules ready to be plugged, without the effort of converting them into DLL conforming code.

Part III

4 IQS main data structures (iqs.h)

The data types, constants, macros and variables of the IQS module can be divided in two complementary subsets:

- the one automatically generated by Lex & Yacc, scattered among **lexyy.c**, **ytab.h** and **ytab.c** files (this set is exclusively related with the Lexical Analysis and Parsing functionalities);

- the other, supporting the search and retrieval of information from the repository, as well as the information exchange between the Visual Basic Interface Layer and the C Layer.

Concerning the Lex & Yacc related subset, all the relevant issues were previously discussed in **2 IQL Parsing** and **3 Importing the generated code to a Windows DLL**.
The focus will now be on the main implementation issues of the second sub-set, whose data type definitions mainly lie at the **iqs.h** header file.

4.1 Handling the repository information (the Set approach)

Basically, the repository search functions being part of the IQS API (yet to be discussed) will try to recover *Sets* of objects which obey to some common properties. Therefore, an implementation of the *Set* Abstract Data Type is opportune.
Such an implementation should provide for:

- *Convenience* - easy access and management of the the physical layout;

- *Efficiency* when performing basic operations over instances of the Set data type, such as creating a set, writing on it, reading it, destroying it, etc.

These are inherently conflicting issues, inevitably introducing some implementation compromises. The next two sections will turn on some light over these subjects.

4.1.1 Dynamic Arrays implementing Sets

Dynamic Arrays are one possible layout for the Set Abstract Data Type.
In the C programing language, implementation of the Dynamic Array concept is straightforward[15]:

[15]This type definition conforms to the Dynamic Array abstract concept established in **7 IQS data structures design** on [IQS-2.1].

```
typedef struct {
   long index;  /* index >=0 means index+1 objects are presently in the array */
                /* index <=-1 means array is empty                            */
   void *array; /* generic pointer to a memory block containing the array     */
} Dynamic_Array;
```

Thus, the `array` field is discarded whenever the `index` field ≤ -1, that is, the pointer is assumed no to be tight with an allocated memory block.

Instead of using the `Dynamic_Array` data type as the unique type for all Sets used by IQS, it was decided to define as much data types as different kinds of Sets one could need (in the IQS context, naturally). This was intended to increase code readability, not only during implementation but also whenever maintenance is needed.

The next is an extract from the **iqs.h** file, with the type definition for all kinds of Sets used by IQS:

```
// a set of ObjIDs
typedef struct {
    long int index;
    ObjID FAR *objids;
} ObjID_list;

// a set of AOIDs
typedef struct {
    long int index;
    AOID FAR *aoids;
} AOID_list;

// a set of names (of attributes, facets, links, etc)
typedef struct {
    long int index;
    char FAR *names;
} Name_list;

// a set of sets of names (of attributes, facets, links, etc)
typedef struct {
    long int index;
    Name_list FAR *list;
} Name_list_list;

// a set of values (of attributes, facets, links, etc);
// main diference between Name_list is the conceptual distance field
typedef struct {
    long int index;
    int distance;
    char FAR *info;
} Array_list;
```

```
// a resolved query is a pair (querytext, queryaoids) with
// queryaoids being a set of type AOID_list
typedef struct {
    char FAR * querytext;
    AOID_list queryaoids;
} ResolvedQuery;

// an History is a set of resolved queries
typedef struct {
    long int index;
    ResolvedQuery FAR * queries;
} Query_list;
```

Dynamic Arrays implemented this way have *pros* and *cons*:

• **main advantage**: provide for a clean and easy implementation of the IQS API (*convenience*); remember that accessing the information field is the same as accessing a static array: a direct access can be made; no tricky linked-list (or even more complex data types) access and maintenance operations are involved;

• **main disadvantage**: whenever the information field needs to be expanded, all the field must be reallocated; this can be a serious drawback if reallocation is an often operation (poor *efficiency*); memory fragmentation could also be an obstacle to reallocation: there could simply do not exist a contiguous memory block big enough to support the reallocation, but the sum of all the tiny free blocks scattered in memory could provide the amount of space needed (linked-lists are a more robust implementation in these circumstances).

Trying to avoid the overhead resulting from repeatedly reallocating a memory block by small pieces, the IQS API used some trivial solutions: in many circumstances a block known to be big enough to cope with all the demands is allocated; other times, if reallocation is really necessary, a big block is requested, trying to delay the next reallocation.

4.1.2 A basic Set API

Given the Dynamic Array based type definitions for all kinds of *Sets* IQS can use, one can easily set up a small package of functions implementing the most common set operations.

Every function should be able to work over different Sets, that is, it should provide for a basic level of *polymorphism*. In C, a possible way to achieve it is to receive parameters `void*` (or `char*`) typed and then, based on a special parameter, a `set_type` integer code, to make the appropriate casts. Every function in the Set API is based in this principle, and therefore has the next basic internal structure:

```
int aSetOperation (void FAR *one_set, void FAR *other_set, int set_type)
{

 switch (set_type) {

 case TYPE1 : /* apply TYPE1 cast to parameters and perform operation on them */
              break;
...
 case TYPEn : /* apply TYPEn cast to parameters and perform operation on them */
              break;
 }

}
```

Not every set operations are defined for all of the *Set* types in **iqs.h**, because some of them never needed those operations during the IQS implementation process (in fact, all operations emerged as they were needed and to extend -if necessary- the actual functionalities to other Set types should be a trivial task). Therefore, in the Set API description the "`type` possible values" item will show the Set types for which the operation being described is defined (in that context, the Set types are integer constants defined in **iqs.h**).

The "secondary effects" item explains what happens to the in-out parameters when the function does not return `IQS_SUCCESS`.

The Set API follows:

```
void iqsCleanSet(void FAR *set, int type)
```

This function inspects the `index` field to check if is greater than -1. If so, it deallocates the information field and resets `index` to -1. Otherwise, leaves the `set` structure intact.

`type` possible values:
- `IQS_OBJIDLIST`
- `IQS_AOIDLIST`
- `IQS_ARRAYLIST`
- `IQS_NAMELIST`
- `IQS_NAMELISTLIST`
- `IQS_QUERYLIST`

```
int iqsCopySet(void FAR *source, void FAR *destiny,
               int type)
```

This function makes `destiny` a copy of `source`. Primitive contents of `destiny` are always cleaned and `source` left intact.

`type` possible values:

- `IQS_AOIDLIST`
- `IQS_ARRAYLIST`
- `IQS_NAMELIST`

Return values:

- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_SUCCESS` - operation successful.

Secondary effects:

- `IQS_NOMEMORY` - cleans `destiny`.

```
int iqsSetDifference(void FAR *a, void FAR *b,
                     int type)
```

Perform the set difference between `a` and `b`, placing the final result in `a`, that is, `a=a-b`.

`type` possible values:

- `IQS_AOIDLIST`
- `IQS_ARRAYLIST`
- `IQS_NAMELIST`

Return values:

- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_SUCCESS` - operation successful.

Secondary effects:

- `IQS_NOMEMORY` - cleans `a`.

```
int iqsSetIntersection(void FAR *a, void FAR *b,
                       int type)
```

Makes `a` the intersection set between `a` and `b`. The intersection is made at the cost of `iqsSetDifference` because `aÇb=a-(a-b)`.

`type` possible values:

- `IQS_AOIDLIST`
- `IQS_NAMELIST`

Return values:

- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_SUCCESS` - operation successful.

Secondary effects:

- `IQS_NOMEMORY` - cleans `a`.

```
int iqsSetUnion(void FAR *a, void FAR *b, int type)
```

Performs `a=aÈb.`

`type` possible values:
- `IQS_AOIDLIST`
- `IQS_NAMELIST`

Return values:
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_SUCCESS` - operation successful.

Secondary effects:
- `IQS_NOMEMORY` - cleans `a`.

```
int iqsMakeSet (void FAR *bag, void FAR *set,
                int type)
```

Makes `set` a copy of `bag` but without repeated objects.

`type` possible values:
- `IQS_AOIDLIST`
- `IQS_ARRAYLIST`
- `IQS_NAMELIST`
- `IQS_NAMELISTLIST`

Return values:
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_SUCCESS` - operation successful.

Secondary effects:
- `IQS_NOMEMORY` - cleans `set`.

4.2 The *ParserState* data type

All the relevant data structures defining, at a precise moment, the state of the IQS querying process, are kept in a record[16]. Gathering that information into a unique, well known, C structure, allows for a better control over the IQS state because when the state changes, one expects to see the changes reflected only in that structure. To verify the present IQS state it is enough to check the data contained in that structure.

[16]see also **7 IQS data structures design** at [IQS-2.1].

The `ParserState` data type implements this view and the variable `iqsState` is a (unique) global instance of that type. This structure encompasses Interface related items as well as Parser related ones. However, nor the Interface neither the Parser is exclusively controlled by its contents:

- Visual Basic Interface control and management details do not cross the Visual Basic/C frontier; `ParserState` only has some fields with the contents of some Interface objects (as list panes, for instance); it also keeps the enable-disable values for the majority of the buttons of the Interface. The semantic actions of the Parser are responsible for keeping these fields with the right contents, conforming the deterministic automata embedded in the Parser; the only thing Visual Basic has to do when Parser Layer returns control, is to call appropriate functions to recover the values of some critical fields and to refresh the Interface accordingly.

- obviously, Lex & Yacc generated data structures controlling Lexical Analysis and Parsing activities are left intact among the generated code; in fact, none of the fields of `ParserState` controls the parsing activity, being instead a direct reflection of the semantic actions; in that sense, the very internal state of the (Lexer, Parser) pair is ignored; only the "external" state, resulting from the internal operations is kept in `ParserState`.

Figure 5 depicts the `ParserState` data type declaration, extracted from **iqs.h**. The meaning of each field follows:

- `AOID_list iqsQS`: *query* state (the set of objects presently solving the *query*);

- `AOID_list iqsQSC`: compounds state (a set of objects being compounds);

- `Array_list iqsQSV`: a set of generic attribute values, or class attributes[17] values, or facet values; in Assisted Mode, these will be the contents of the interface list-pane presenting the available values to chose from, depending on having previously selected a generic attribute or a class attribute or a facet, respectively in another list-pane (refer to `iqsAOGAttribs`, `iqsCLAAttribs` and `iqsFACAttribs` fields description);

- `Name_list_list iqsListList`: a set in which each element is himself another set; to fully understand the need for this field refer to the IQS API description of the `iqsGetClassAttributes`, `iqsGetSourcesAndLinks` and `iqsGetSourcesAndLinksBySinks` functions, on section **5 The IQS API**.

- `char FAR * iqsQuery`: this field matches, at every moment, the part of the query text already parsed (and thus solved); in Batch Mode, both this field and the complete query phrase being solved will be the same[18] at the end of the

[17]also called class attributes

[18]except that `iqsQuery` will always have IQL tokens uppercased (this does not refer to identifiers, however); also, in the case of the query being derived from a non-kernel template, references to other queries may have been converted from local to global ones (recall **6 The History** at [IQS-2.1]).

```
typedef struct {
    AOID_list iqsQS;
    AOID_list iqsQSC;
    Array_list iqsQSV
    Name_list_list iqsListList;
    char FAR * iqsQuery;
    Query_list iqsQueryHistory;
    VBState iqsVBState;
    Name_list iqsAOGAttribs;
    Name_list iqsAOGAttribsPrevious;
    Name_list iqsFACAttribs;
    Name_list iqsFACAttribsPrevious;
    Name_list iqsCLAAttribs;
    Name_list iqsCLAAttribsPrevious;
    Name_list iqsSLCNames;
    Name_list iqsPHANames;
    Name_list iqsPHANamesPrevious;
    Name_list iqsCRLNames;
    Name_list iqsCRLNamesPrevious;
    Name_list iqsLNKNames;
    bool AOGExplored;
    bool FACExplored;
    bool CLAExplored;
    bool SLCExplored;
    bool PHAExplored;
    bool IsCompoundPressed;
    bool CRLExplored;
    bool BelongToCompoundPressed;
    AOID_list iqsRM;
    BOOL iqsBatchOn;
    long int iqsNextLocalHIndex;
} ParserState;
```

Figure 5 -- the `ParserState` data type declaration

resolution process; also, at the end of the query solving, in Assisted Mode, `iqsQuery` will contain the submitted query phrase after redundancy and incompleteness[19]have been purged; so, whatever mode of operation considered, an internal synthesized query will be kept at `iqsQuery` and, in the case of a successful resolution, it will be added to the History, together with the objects which were the query solution, contained by the `iqsQS` field;

- `Query_list iqsQueryHistory`: this field stands for the History of the present IQS session; refer to section **4.2.1** for a complete description of all the related implementation details;

- `VBState iqsVBState`: this field keeps the enable/disable state of all the Visual Basic layer buttons (as well as some list boxes and menus) under the control of

[19]recall **2.4.1 Aided Mode IQL sub-grammar issues**.

the deterministic automata of the IQL grammar; to get a more detailed description of these field, refer to section **4.2.2**;

- `Name_list iqsAOGAttribs`, `Name_list iqsFACAttribs`, `Name_list iqsCLAAttribs`, `Name_list iqsSLCNames`, `Name_list iqsPHANames`, `Name_list iqsCRLNames`, `Name_list iqsLNKNames`: these fields are the sets of generic attributes, facets, class attributes, software life cycles, phases, characteristic relations and links, respectively, available to be chosen from a dedicated list-pane[20];

- `bool AOGExplored`, `bool FACExplored`, `bool CLAExplored`, `bool SLCExplored`, `bool PHAExplored`, `bool CRLExplored`: these flags are enabled when the refinement by generic attributes, facets, class attributes, software life cycles, phases or characteristic relations, respectively, is found to be finished; in this situation, preventing further attempts to refine by these paths is done by inspecting the respective flags;

- `bool IsCompoundPressed`, `bool BelongToCompoundPressed`: these are flags enabled whenever the respective buttons are pressed; when that happens, those buttons will not ever be allowed to be enabled again because the associated action can be performed only once.

- `AOID_list iqsRM`: the only purpose of this field is to receive a copy of a temporary or final query solution, kept by `iqsQS`, in order to let appropriate functions[21] handle those results and submitting them to the Result Manager;

- `BOOL iqsBatchOn`: this is a flag activated when IQS enters the Batch Mode; mainly, this flag allows for flow control inside the semantic actions shared code, preventing Assisted Mode specific code to be executed.

- `long int iqsNextLocalHIndex`: this field keeps track of the next available local index during some History operations; his usefulness is fully explained at section **4.2.1**

Having specific fields to keep objects that are compounds (`iqsQSC`) or to be used by Result Manager related operations (`iqsRM`), does not necessarily mean that during every query resolution, their contents are meaningful. That depends on the *Template* format of the query presently being solved. This also applies to the flags `AOGExplored`, `FACExplored`, `CLAExplored`, `SLCExplored`, `PHAExplored`, `CRLExplored`, `IsCompoundPressed` and `BelongToCompoundPressed`, essentially related with visual features used only at some specific *Templates*.

4.2.1 Implementation details of the History

The C data types implementing the History abstract definition provided at chapter **6** of [IQS-2.1] are[22]:

[20]see also section **4.2.2 The VBState data type**.
[21]see section **7 The IQS Visual Basic related API**.
[22]see also section **4.1.1 Dynamic Arrays implementing Sets**.

```
// a solved query is a pair (querytext, queryaoids) with
// queryaoids being a set of type AOID_list
typedef struct {
    char FAR * querytext;
    AOID_list queryaoids;
} ResolvedQuery;

// an History is a set of resolved queries
typedef struct {
    long int index;
    ResolvedQuery FAR * queries;
} Query_list;
```

As section **4.1.1** already referred, these data types are extensions to the `Dynamic_Array` basic Set data type. Also, as **4.2** showed, the field `iqsQueryHistory` of the `ParserState` structure implements the IQS History.

One important issue concerning the management of the present IQS History is the one related with adding to it previously solved queries, kept in a saved History. This possibility has been envisioned at the chapter **6** of [IQS-2.1], which even described what had to be done in order to have the local references made by non-kernel queries, at the imported History, to become global ones, at the global[23] History.

On the basis of that intended behaviour, the next three macros are used to maintain the local and global references aside:

- `#define iqsGetNextGlobalHIndex() (iqsState.iqsQueryHistory.index+1)`

  This macro retrieves the next valid global index (or reference), that is, the next "vacancy" on the History. Every time a query is successfully solved, it is added to the History, and his text will have been prefixed with `#NUMBER` where `NUMBER` will be the next global index value, as given by the macro. This is true even for those queries coming from an imported History (his text already contained a local prefix, which however will only be used in local references).

- `#define iqsSetNextLocalHIndex(index) iqsState.iqsNextLocalHIndex=index`

  This macro is called to set the `iqsNextLocalHIndex` field of `iqsState` to zero, every time a batch resolution is started. The Batch Mode always defines a local context, no matter the state of the global History and thus needs the proper index prefix on every query phrase. On the opposite side, the Assisted Mode always operates, by default, on a global context because it does not require the users to provide for an index to the query being interactively solved, instead choosing the next global valid one.

[23]that is, the History resulting from joining the loaded and the present one.

- `#define iqsGetNextLocalHIndex() (iqsState.iqsNextLocalHIndex)`

In Batch Mode, every time a query phrase recognition starts, the mandatory index prefix[24] is checked to see if it matches the one expected in that local context, given by this macro.

4.2.2 The VBState data type

In Assisted Mode, besides providing for the contents of the various Visual Basic objects presented at the interface layer, the underlying C layer also must enable and disable them. In fact, that is how the C code implementing the IQL grammar parsing mechanism, manages to control, in a deterministic way, the interface.

Figure 6 presents the C data type definition of the `VBState` structure, containing the necessary fields to control all the relevant *interface* objects on Assisted Mode. The meaning of each field is also explained.

```
typedef struct {
    BOOL outClasses;  /* enable/disable the class hierarchy tree       */
    BOOL cmdGEN;      /* enable/disable the generic attributes button  */
    BOOL cmdFAC;      /* enable/disable the facets button              */
    BOOL cmdATT;      /* enable/disable the class attributes button    */
    BOOL lstAttrName; /* enable/disable the list-pane showing generic  */
                      /* attributes, facets, class attributes, software*/
                      /* life cycles, phases, characteristic relations,*/
                      /* and link names                                */
    BOOL lstAttrValue;/* enable/disable the list-pane showing generic  */
                      /* attributes, facets or class attributes values */
    BOOL cmdCRL; /* enable/disable the characteristic relations button */
    BOOL cmdISC;      /* enable/disable the is-compound button         */
    BOOL cmdBLC;      /* enable/disable the belong-to-compound button  */
    BOOL cmdCHECK;    /* enable/disable the check button               */
    BOOL cmdABORT;    /* enable/disable the abort button               */
                      /* the next eight buttons enable/disable the     */
                      /* access to a specific template */
    BOOL cmdT1;
    BOOL cmdT2;
    BOOL cmdT3;
    BOOL cmdT4;
    BOOL cmdT5;
    BOOL cmdT6;
    BOOL cmdT7;
    BOOL cmdT8;
    BOOL lstHistory;  /* enable/disable access to history visualization*/
    BOOL cmdSLC;      /* enable/disable the software life cycles button*/
    BOOL cmdPHA;      /* enable/disable the phases button              */
    BOOL spnFAC;   /* enable/disable the conceptual distance scrollbar */
    BOOL mnuHST;      /* enable/disbale access to the History menu     */
    BOOL mnuDSP;      /* enable/disbale access to the Display menu     */
} VBState;
```

Figure 6 - The VBState data type

[24] of the format `#NUMBER`.

Note that the enabling and disabling of the `VBState` items takes place at the body of the semantic actions code, because being embedded in the IQL grammar, these ones are context sensitive and thus know exactly which visual items to enable or to disable, at any step of the query recognition process.

5 The IQS API (iqs.c)

Until now, we have refered to the IQS API as the set of functions callable from the semantic actions code and exclusively concerned with retrieving objects from the repository, by invoking the appropriate functionalities of the SOURLIB software layer, during query resolution.

However, in practical terms, the C code file implementing the IQS API, **iqs.c**, includes also other sets of functions, some of them offering services to the IQS API functions, and others making possible to the objects collected from the repository by the IQS API to access the interface upper layer and even to control its behaviour. These other functionalities group themselves into three distinct small sets, inside the **iqs.c** file:

- a basic *Set* API, already presented at **4.1.2**;
- an Auxiliary IQS API, to be discussed at **5.1**;
- an IQS Visual Basic related API, whose description is postponed until chapter **7**;

The following is a description of the IQS API, similar to the one provided at **Appendix B** of [IQS-2.1]. Note the "Secondary effects" field, which explains what happens to the in-out parameters when the function returns some specific values (generally all but `IQS_SUCCESS`[25]).

```
int iqsGetHierarchyAoids(AOID_list FAR *aoid_list,
                         char FAR *class)
```

Given a `class` name, this function puts in `aoid_list` all the AOIDs of the class sub-hierarchy starting at `class`. Based on `eraGetObj` calls for each class bellow the one provided, `iqsGetHierarchyAoids` will remove, from `aoid_list`, the *system-object* TUTTO (used by Comparator-Modifier as a upper-bound to close the lattice - see [CM-1.4 1993]), if found during the search.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no objects found for the selected class hierarchy;
- `IQS_SUCCESS` - operation successful.

[25]as already defined at **4.1.2 A basic Set API**.

Secondary effects:
• `IQS_PARAMERR`, `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `aoid_list`

```
int iqsGetAOGAttribs (AOID_list FAR *aoid_list,
                      Name_list FAR *aog_attrs)
```

`iqsGetAOGAttribs` will search the generic attributes for whom the objects in `aoid_list` define a value, that is, for each object in `aoid_list`, the "`AOG`" class is inspected via `eraGetObject` in order to check if each generic attribute has a well-defined non-empty value. As soon as a value has been found for all the generic attributes, the search is stopped (this could happen at the very first object of `aoid_list` if this object defines a non-empty value for all of the generic attributes). The defined generic attributes (except "`AOID`") are returned via the *in/out* parameter `aog_attrs`.

Return values:
• `IQS_PARAMERR` - bad parameters; operation aborted;
• `IQS_NOMEMORY` - not enough memory; operation aborted;
• `IQS_ERROR` - internal or unknown error; operation aborted;
• `IQS_NOTFOUND` - no generic attributes defined (except "`AOID`");
• `IQS_SUCCESS` - operation successful.

Secondary effects:
• `IQS_PARAMERR`, `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `aog_attrs`.

```
int iqsGetAOGValues(AOID_list FAR *aoid_list,
                    Array_list FAR *attr_values)
```

For each object in `aoid_list`, `iqsGetAOGValues` inspects the "`AOG`" class via `eraGetObject`, checking for the value the generic attribute passed in `attr_values->info` assumes. The goal is to make `attr_values` the set of those values.

Return values:
• `IQS_PARAMERR` - bad parameters; operation aborted;
• `IQS_NOMEMORY` - not enough memory; operation aborted;
• `IQS_ERROR` - internal or unknown error; operation aborted;
• `IQS_NOTFOUND` - generic attribute `attr_values->info` unknown;
• `IQS_NOVALUES` - generic attribute `attr_values->info` undefined;
• `IQS_SUCCESS` - operation successful.

Secondary effects:
• IQS_PARAMERR, IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND, IQS_NOVALUES: cleans attr_values.

```
int iqsGetFacets(AOID_list FAR *aoid_list,
                 Name_list FAR *facets)
```

iqsGetFacets will search the facets for whom the objects in aoid_list define a value, that is, for each object in aoid_list, the "FACETS" class is inspected via eraGetObject in order to check if each facet has a well-defined non-empty value. As soon as a value has been found for all the facets, the search is stopped (this could happen at the very first object of aoid_list if this object defines a non-empty value for all of the facets). The defined facets are returned via the *in/out* parameter facets.

Return values:
• IQS_PARAMERR - bad parameters; operation aborted;
• IQS_NOMEMORY - not enough memory; operation aborted;
• IQS_ERROR - internal or unknown error; operation aborted;
• IQS_NOTFOUND - no facets defined;
• IQS_SUCCESS - operation successful.

Secondary effects:
• IQS_PARAMERR, IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND: cleans facets.

```
int iqsGetFacetsValues(AOID_list FAR *aoid_list,
                       Array_list FAR *facet_values)
```

For each object in aoid_list, iqsGetFacetsValues inspects the "FACETS" class via eraGetObject, checking for the value the facet in facet_values->info assumes. The goal is to make facet_values the set of those values.

Return values:
• IQS_PARAMERR - bad parameters; operation aborted;
• IQS_NOMEMORY - not enough memory; operation aborted;
• IQS_ERROR - internal or unknown error; operation aborted;
• IQS_NOTFOUND - facet facet_values->info unknown;
• IQS_NOVALUES - facet facet_values->info undefined;
• IQS_SUCCESS - operation successful;

Secondary effects:
• IQS_PARAMERR, IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND, IQS_NOVALUES: cleans facet_values.

```
int iqsGetClassAttributes(AOID_list FAR *aoid_list,
                Name_list_list FAR *class_atts_list)
```

`iqsGetClassAttributes` will search the class attributes for whom the objects in `aoid_list` define a value, that is, for each object in `aoid_list`, the "`AOG`" class is inspected via `eraGetObject` in order to retrieve the value of the "`CLASS`" generic attribute; the class whose name is given by that value is then inspected, once again using `eraGetObject`, and all its attributes, having a well-defined non-empty value, are retrieved into a set of names; this set is object specific and so this task must always be done for every object of `aoid_list`. Since a set of class attributes is eventually needed for each object, the *in/out* parameter, `class_atts_list`, is a set of set of names.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no class attributes defined;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
- `IQS_PARAMERR`, `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `class_atts_list`.

```
int iqsGetAttribsValues(AOID_list FAR *aoid_list,
                       Array_list FAR *attr_values)
```

For each object in `aoid_list`, `iqsGetAttribsValues` inspects the "`AOG`" class via `eraGetObject`, checking for the value of the "`CLASS`" generic attribute; the class whose name is given by that value is then inspected, once again using `eraGetObject`, in order to retrieve the value of the class attribute originally contained in `attr_values->info`. The goal is to make `attr_values` the set of the values obtained that way.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - class attribute `attr_values->info` unknown;
- `IQS_NOVALUES` - class attribute `attr_values->info` undefined;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
- `IQS_PARAMERR`, `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`, `IQS_NOVALUES`: cleans `attr_values`.

```
int iqsGetSLCs(AOID_list FAR *aoid_list,
               Name_list FAR *slcs)
```

For each object in `aoid_list`, `iqsGetSLCs` inspects the "`PRJ`" class via `eraGetObject`, checking for a well-defined non-empty value of the "`SLC`" (Software Life Cycle) attribute. At the end, `slcs` will contain the Software Life Cycles retrieved that way.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no software life cycles defined;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
- `IQS_PARAMERR`, `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `slcs`.

```
int iqsGetPHAs(AOID_list FAR *aoid_list,
               Name_list FAR *phas,
               Name_list FAR *slcs)
```

Firstly, `iqsGetSLCs` is called in order to get into `slcs` the Software Life Cycles of the `aoid_list` objects. After that, `iqsGetPHAsBySLC` will check, for each Software Life Cycle, his specific Software Life Cycle Phases. At the end, `phas` will contain the Software Life Cycles Phases retrieved that way.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no software life cycles or no phases defined;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
- `IQS_PARAMERR`, `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `phas` and `slcs`.

```
int iqsGetPHAsBySLC(Name_list FAR *phas_list,
                    char FAR *slc)
```

Given a Software Life Cycle `slc`, `conGetSLCPHA` is invoked in order to retrieve all the Software Life Cycle Phases of that Software Life Cycle into `phas_list`.

Return values:

- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no phases found for the software life cycle `slc`;
- `IQS_SUCCESS` - operation successful;

Secondary effects:

- `IQS_PARAMERR, IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `phas_list;`

```
int iqsGetAoidsBySLC(AOID_list FAR *aoid_list,
                     char FAR *slc)
```

For each object in `aoid_list`, `iqsGetAoidsBySLC` inspects the "`PRJ`" class via `eraGetObject`, checking for the value of the "`SLC`" (Software Life Cycle) attribute. At the end, `aoid_list` will keep only the objects for whom the value of the "`SLC`" attribute equals the `slc` parameter.

Return values:

- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOAOIDSSLCS` - no objects found with any software life cycle;
- `IQS_NOAOIDSSLC` - no objects found with `slc`;
- `IQS_SUCCESS` - operation successful;

Secondary effects:

- `IQS_PARAMERR, IQS_NOMEMORY`, `IQS_ERROR, IQS_NOAOIDSSLCS, IQS_NOAOIDSSLC`: cleans `aoid_list`.

```
int iqsGetSLCsByPHA(Name_list FAR *slcs_list,
                    char FAR *pha)
```

For each Software Life Cycle in `slcs_list`, calls `iqsGetPHAsBySLC` retrieving all its Phases. Then, it checks if `pha` is among those Phases. At the end, `slcs_list` will keep only those Software Life Cycle *containing* `pha`.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no software life cycles found with `pha`;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
- `IQS_PARAMERR, IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `slcs_list;`

```
int iqsGetCompounds(AOID_list FAR *aoid_list)
```

For each object in `aoid_list`, `iqsGetCompounds` calls `conGetMbr` once, verifying if it returns `A_SUCCESS`, in wich case the object is assumed to be a compound object. At the end, `aoid_list` will keep only those objects which passed the previous test, that is, those objects being compounds.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no compounds found;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
- `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `aoid_list`.

```
int iqsGetCaractRel(AOID_list FAR *aoid_list,
                    Name_list FAR *crls)
```

For each object in `aoid_list`, `iqsGetCaractRel` calls `conGetMbrLnk` in order to retrieve a set of `ObjIDs`, each one standing for a Characteristic Relation. `conGetLnk` will then allow for each one of those `ObjIDs` to be maped into a string:

the name of the Characteristic Relation. In the end, `crls` will contain the set of Characteristic Relation names retrieved as described.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no characteristic relations found;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
- `IQS_PARAMERR`, `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `crls`.

```
int iqsGetClustersByCaractRel(
                            AOID_list FAR *aoid_list,
                            char FAR *crl)
```

For each object in `aoid_list`, `iqsGetClustersByCaractRel` calls `conGetMbrLnk` in order to retrieve a set of `ObjIDs`, each one standing for a Characteristic Relation. `conGetLnk` will then allow for each one of those `ObjIDs` to be maped into the name of the respective Characteristic Relation. If the parameter `crl` matches at least one of these Characteristic Relations, then the object currently under survey is considered to be a Cluster (being `crl` one of his Characteristic Relation). In the end, `aoid_list` will keep only the objects being Clusters.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no clusters found with any Characteristic Relation;
- `IQS_NOVALUES` - no clusters found with the Characteristic Relation `crl`;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
- `IQS_PARAMERR`, `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`, `IQS_NOVALUES`: cleans `aoid_list`.

```
int iqsGetClaoByMember(AOID_list FAR *aoid_list)
```

The objects that aggregate the ones in `aoid_list`, are retrieved and placed there. `conGetMbr` is the low-level functionality on which `iqsGetClaoByMember` mainly relies for that purpose.

Return values:
- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or uknown error; operation aborted;
- IQS_NOTFOUND - no compounds found;
- IQS_SUCCESS - operation successful.

Secondary effects:
- IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND: cleans aoid_list.

```
int iqsGetMemberByClao(AOID_list FAR *aoid_list)
```

For each object in aoid_list, iqsGetMemberByClao calls conGetMbr, retrieving all his members. At the end, aoid_list will be the set of all the objects contained by the ones initialy there.

Return values:
- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_NOTFOUND - no compounds found;
- IQS_SUCCESS - operation successful;

Secondary effects:
- IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND: cleans aoid_list.

```
int iqsGetSources(AOID_list FAR *aoid_list)
```

For each object in aoid_list, iqsGetSources calls conGetLnk, in order to check if the current object is source of some link. At the end, aoid_list will keep only the source objects.

Return values:
- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_NOTFOUND - no sources found;
- IQS_SUCCESS - operation successful;

Secondary effects:
- IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND: cleans aoid_list.

```
int iqsGetSourcesAndLinks(AOID_list FAR *aoid_list
                    Name_list_list FAR *links_set_list)
```

For each object in `aoid_list`, `iqsGetSourcesAndLinks` calls `conGetLnk`, in order to check if the current object is source of a link. If so, the set of all the outgoing links from that object is retrieved. At the end, `aoid_list` will keep only the source objects and `links_set_list` will contain the respective sets of outgoing links.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no sources found;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
- `IQS_PARAMERR`: cleans `links_set_list`;
- `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `aoid_list` and `links_set_list.`

```
int iqsGetSourcesByLinkAndSinks(
                            AOID_list FAR *aoid_list,
                            char FAR *link,
                            AOID_list FAR *sinks)
```

For each object in `aoid_list`, `iqsGetSources` calls `conGetLnk`, in order to check if the current object is source of `link` to at least one sink in `sinks.` At the end, `aoid_list` will keep only the objects founded to be sources in this way.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no sources found;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
- `IQS_PARAMERR, IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `aoid_list` and `sinks.`

```
int iqsGetSourcesAndLinksBySinks(
                  AOID_list FAR *aoid_list,
                  Name_list_list FAR *links_set_list,
                  AOID_list FAR *sinks)
```

For each object in `aoid_list`, `iqsGetSources` calls `conGetLnk`, in order to check if the current object is source of some link to some sink in `sinks`. If so, the set of all the outgoing links from that object to all the `sinks` is retrieved. At the end, `aoid_list` will keep only the source objects and `links_set_list` will contain the respective sets of outgoing links to at least one of the `sinks`.

Return values:

- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no sources found;
- `IQS_SUCCESS` - operation successful;

Secondary effects:

- `IQS_PARAMERR, IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `aoid_list` and `sinks.`

5.1 An Auxiliary IQS API

This section describes auxiliary functions developed to handle some low-level implementation aspects (otherwise, IQS API functions would have, internally, to deal with them), namely:

- safe memory reallocation (mostly done to expand sets of objects);

- token retrieving from token strings based on the separator `','`.

These details should be hided from the majority of the IQS API functions to take care of them. Besides encapsulation, having functions to perform very common low-level tasks allowed a faster implementation of the IQS API.

The next is a description of the Auxiliary IQS API:

```
int iqsFrealloc (void FAR **memptr, size_t memsize,
                 int ptrtype)
```

This function reallocates a memory block. It receives the address of a `void FAR *` pointer, - `memptr -`, the new intended size (in bytes) of the memory block tight with `memptr`, - `memsize -`, and an integer - `ptrtype` -, coding the pointer type in order to make appropriate internal casts. `iqsFrealloc` is based on a call to

```
void FAR * _frealloc (void FAR * memblock, size_t size)
```

of the `malloc.h`, Microsoft Visual C++ 1.5 library, and intends to avoid the loosing of the pointer in reallocation, if `_frealloc` returns `NULL` and a backup of the previous contents of the pointer has not been made.

`ptrtype` possible values:

- `IQS_AOIDFARPTR`
- `IQS_CHARFARPTR`
- `IQS_NAMELISTFARPTR`
- `IQS_OBJIDFARPTR`
- `IQS_RESOLVEDQUERYFARPTR`

Return values:

- `IQS_NOMEMORY` - not enough memory; operation aborted (`*memptr` remains intact);
- `IQS_SUCCESS` - operation successful (`*memptr` now points to the reallocated memory block);

```
char FAR * iqsStrtok (char FAR *str)
```

Enhance the functionality of the `_fstrtok` function of the `string.h` Microsoft Visual C++ 1.5 library:

- returns, one by one, the tokens in `str` even if it contains empty tokens; remember that `_fstrtok` would simple ignore them; however, only the character `','` (coma) is considered to be a token delimiter;

- does not destroy the contents of `str` because it operates on an internal copy, while `_fstrtok` overwrites the token delimiter with a `'\0'` character every time it finds a token.

`iqsStrtok` follows the same invocation policy as `_fstrtok`: at the first call, the `str` parameter must not be a `NULL` pointer, and the first token found is returned; next calls will have to be made precisely with a `NULL` pointer in order to retrieve the rest of the tokens; the function returns tokens, one by one, on successive calls; once it does not find more tokens it will always return NULL.

Note that an empty token returned is a `(char FAR *)""`, that is, an empty string. This is not the same as a `(char FAR *)NULL` which means that the function cannot find more tokens in the string. For instance, the strings `""` (empty string)`,` "," and "hello," would make `iqsStrtok` to return, respectively, `""` and `NULL`, `""` and `""` and `NULL`, "hello" and `""` and `NULL`.

```
int iqsCheckWordInList(char FAR *word, char FAR *list
                       ,int FAR *index, int mode)
```

This function is based on calls to `iqsStrtok`, providing different functionalities, accordingly with the parameter `mode`. It has been specifically implemented to extend `iqsStrtok` capabilities in handling token retrieving over strings where the delimiter is the character `','` (coma).

`mode` possible values:

• `IQS_WORD`: get the `index`th token in `list` and return it via `word`;
• `IQS_STARTINDEX`: search for the first occurrence of `word` in `list` and get its relative position into `index`; if `word` does not exist, then, after `iqsCheckWordInList` returns, the total number of tokens contained within `list` (including empty tokens - `""`-) will be equal to `index+1`;
• `IQS_NEXTINDEX`: like `IQS_STARTINDEX`, but used to retrieve the indexes of `word` beyond the first occurrence in `list`; it can only be used after first invoking `iqsCheckWordInList` with the `IQS_STARTINDEX` mode.

Return values:
• `IQS_PARAMERR` - bad parameters; operation aborted;
• `IQS_NOTFOUND` - no more tokens found at `list`;
• `IQS_SUCCESS` - operation successful;

```
int iqsGetAttrValue(char FAR *names, char FAR *name,
                    char FAR *values, char FAR *value)
```

This function is based on calls to `iqsCheckWordInList` and it searches for the index[26] of `name` in `names` and then for the correspondent `value` in `values` (the correspondent `value` is the one with a relative position within `values` equal to the relative position of `name` in `names`). `iqsGetAttrValue` is almost exclusively used to make the "projection" of the `AttrName` field (of the `eraAttrName` ERA structure data type), over the correspondent `AttrValue` list, in order to get a specific pair *(name, value)*. If the retrieved `value` is a whitespace character string, it is converted into an empty string.

Return values:
• `IQS_PARAMERR` - bad parameters; operation aborted;
• `IQS_NOTFOUND` - `name` or `value` could not be found in `names` or `values`, respectively;
• `IQS_SUCCESS` - operation successful.

6 The IQS Semantic Actions API (actions.c)

This chapter presents the C functions implementing the Semantic Actions in the **iqs.y** IQL grammar description. Besides the Semantic Actions code, the file **actions.c** also contains a set of Semantic Actions related Auxiliary functions, which will be described at **6.1**.

Remember that the Semantic Actions will be the primary functions responsible for query resolution and indirect interface control. They are based on calls to the IQS API (in charge with getting from the SOURLIB functionalities the desired repository objects) as well as to their auxiliary functions.

[26]or relative position.

All Semantic Actions are `void` functions, using a global `int` variable, `rIQS`[27] to "return" their results. This has to do with the need of testing the result from the `yyparse` parsing function independently of the result of the semantic action just executed. Therefore, it was decided to keep the return value of `yyparse` to reflect the success or failure of the lexical analysis and parsing activities, and to rely on `rIQS` to know how the semantic actions terminated. This avoids changing the generated code which implements `yyparse`, in order to introduce `return(rIQS)` statements at the proper places[28].

Therefore, instead of having a topic named "Return values:", the IQS Semantic Actions API description that follows, uses, alternatively the "Return values (rIQS):" item. Also, the behaviour of a certain function may vary slightly from Assisted Mode to Batch Mode and so two descriptions are given, one for each operation mode[29]. Note that in the Assisted Mode description, the mentioned tokens of a query phrase enter that phrase as a result of an interface event but in Batch Mode a complete textual description (containing those tokens) is assumed to be provided at once.

`void iqsSAcheck()`

Assisted Mode behaviour:

This function is called whenever the query phrase recognition is considered to be terminated. In Assisted Mode this will happen only explicitly by pressing the `CHECK` interface button and thus introducing the token `CHECK` into the query phrase. `iqsSAauxAddQueryToHistory` is invoked in order to add the query phrase (presently in `iqsState.iqsQuery)` to the History. However, before that, in the case of a query phrase of the `TEMPLATE4` variant, `iqsSAcheck` must check first for the AOIDs previously retrieved (kept in `iqsState.iqsQS`) and belonging to the CLAOs being characterized (and kept in `iqsState.iqsQSC`); this will involve calling the functions `iqsGetMemberByClao` and `iqsSetIntersection`.

Batch Mode behaviour:

In Batch Mode, every time a complete query phrase from the batch is recognized, `iqsSAcheck` is called. The log file of the batch session will be appended with the results of the query just solved and `VBiqsResetParser` will be invoked before processing the next query.

Return values (`rIQS`):

- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_SUCCESS` - operation successful.

---

[27]declared at **iqs.h**.

[28]see also **7 The IQS Visual Basic related API**.

[29]remember that the semantic actions are shared between these two modes, and necessarily certain details will be handled differently inside the same semantic action implementation code.

`void iqsSAabort()`

Assisted Mode behaviour:

This function is callable by pressing the `ABORT` button (and thus making the `ABORT` token to enter the query phrase). `VBiqsResetParser` is invoked in order to abort the query and prepare for the next query solving.

Batch Mode behaviour:

Not callable because a batch resolution terminates only when the last query has been solved or an error occurred.

Return values (`rIQS`):

- `IQS_SUCCESS` - operation (always) successful.

`void batchIqsSAcheckIndex(int index)`

Assisted Mode behaviour:

Not callable. In Assisted Mode the index of a query is automatically associated with that query as soon as the query text begins to be synthesized.

Batch Mode behaviour:

`batchIqsSAcheckIndex` is called every time a token of the format `#index` (where `index` is an integer) is recognized during a batch solving of a query of any *Template* variant. This function checks if the parameter `index` matches the next expected History index in the local context of the present Batch session.

Return values (`rIQS`):

- `IQS_BATCHINDEXNOTVALID` - unexpected index; operation aborted; only in Batch Mode;
- `IQS_SUCCESS` - operation successful.

`void iqsSAinitGetAllClass (int template)`

Assisted Mode behaviour:

This function is called every time one of the interface buttons `#1` to `#4` is pressed (making one of the tokens of the format `TEMPLATEx`[30] to be joined to the query text). `iqsSAinitGetAllClass` will start the internal query phrase synthesis with the text "`#index TEMPLATEx GET ALL CLASS=`", where `index` is returned by `iqsGetNextGlobalHIndex`, and `x`, depending on the `template` parameter, will assume a value among 1 and 4; `iqsSAauxSetVBState` will set next interface state.

Batch Mode behaviour:

[30]x varying from 1 to 4.

Except that `iqsSAauxSetVBState` is not called, the rest of the function acts as in Assisted Mode.

Return values (`rIQS`):
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_SUCCESS` - operation successful.

Secondary effects:
- `IQS_NOMEMORY` - cleans `iqsState.iqsQuery`;

`void iqsSAgetAoidsBellowClass(char FAR * class)`

Assisted Mode behaviour:

This function is called after choosing a `class` in the hierarchy presented at interface level, while making *Template1* to *Template4* query synthesis. `iqsSAgetAoidsBellowClass` will call `iqsGetHierarchyAoids` with `iqsState.iqsQS` and `class` as parameters in order to receive in `iqsState.iqsQS` all the AOIDs bellow `class`. The `class` string will be added to the internal query phrase being synthesized and the next interface state will be set via `iqsSAauxSetVBState`.

Batch Mode behaviour:

Except that `iqsSAauxSetVBState` is not called, the rest of the function behaves as in Assisted Mode.

Return values (`rIQS`):
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no objects found for the selected class hierarchy;
- `IQS_SUCCESS` - operation successful.

`void iqsSAafterAttrTypeChoice(int type)`

Assisted Mode behaviour:

In Assisted Mode, `iqsSAafterAttrTypeChoice` will be called after pressing one of the buttons (coded in the `type` parameter) standing for the generic or class attributes, facets, software life cycles, phases or characteristic relations. The tokens `AOGNAME`, `ATTNAME`, `FACNAME`, `SLCNAME`, `PHANAME` and `CRLNAME` will reflect, at the query phrase, the pressed button. If the refinement is still possible by the way just chosen, then the respective set in `iqsState`, may have to be updated[31], by calling `iqsSAauxInitAttrLists`, in order to provide semantic assistance when choosing

[31]this will only happen if there was a previous refinement and the set has not been updated yet.

later a name or value from that set. `iqsSAauxSetVBState` will set the next interface state.

Batch Mode behaviour:
Always successful because there's no need to assure semantic assistance.

Return values (`rIQS`):
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no objects found with defined generic attributes, or class attributes, or facets, or software life cycles or phases or characteristic relations;
- `IQS_SUCCESS` - operation successful.

```
void iqsSAgetAttrValues(int attrtype,
                        char FAR *attrname)
```

Assisted Mode behaviour:
This function is called in the case of a *Template1* to *Template4* and *Template8* query variants, after choosing one of the possible generic attributes, facets or class attributes (coded in `attrtype`), from an interface list pane with their names. `iqsSAgetAttrValues` will then call `iqsGetAOGValues`, `iqsGetFacetsValues` or `iqsGetAttribsValues`, respectively, in order to fill `iqsState.iqsQSV` with the values specific to those names. `iqsSAauxSetVBState` will set the next interface state.

Batch Mode behaviour:
In Batch Mode, `iqsSAgetAttrValues` will immediately return if the provided attribute or facet (`attrname`) is unknown or no values were found for it. `iqsSAauxSetVBState` is not called; the rest of the function works as in Assisted Mode.

Return values (`rIQS`):
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_BATCHNOGENATTR` - generic attribute unknown; operation aborted; only in Batch Mode;
- `IQS_BATCHNOGENVAL` - generic attribute without values; operation aborted; only in Batch Mode;
- `IQS_BATCHNOFACATTR` - facet unknown; operation aborted; only in Batch Mode;
- `IQS_BATCHNOFACVAL` - facet without values; operation aborted; only in Batch Mode;
- `IQS_BATCHNOATTATTR` - class attribute unknown; operation aborted; only in Batch Mode;

- `IQS_BATCHNOATTVAL` - class attribute without values; operation aborted; only in Batch Mode;
- `IQS_SUCCESS` - operation successful.

```
void iqsSAgetAoidsByValue (int attrtype,
                           char FAR* attrname,
                           char FAR *attrvalue,
                           int condist)
```

Assisted Mode behaviour:

This function is called in the case of a *Template1* to *Template4* and *Template8* query variants, after choosing one of the possible values of a generic attribute, facet or class attribute from the appropriate interface list pane. `iqsSAgetAoidsByValue` starts by updating `iqsState.iqsQSV.distance` with the value of the parameter `condist` (which stands for *conceptual distance*) to be considered if `attrtype` means that one is refining by facets. Then, `iqsSAauxSelectAoidsByValue` is invoked with the `attrvalue` parameter, so that in `iqsState.iqsQS` (or possibly in `iqsState.iqsQSC`) will only remain those objects having the value `attrvalue` for the attribute or facet whose name is `attrname`. If the later task is successful, `attrname` is joined to appropriate (depending on `attrtype`) set of previous chosen names (this will prevent users from choosing later the same generic attribute, facet or class attribute). Also, if `attrname` was the last item of his list, then the `attrtype` refinement path is considered explored, becoming inaccessible. Once filtering of `iqsState.iqsQS` (or `iqsState.iqsQSC`) has successfully occurred, the lists of available generic attributes, facets, class attributes (and possibly software life cycles, phases and characteristic relations) are cleaned, in order to enforce their update if, later, one decides to make a refinement of the same kind. Finally `iqsSAauxSetVBState` and `iqsSAauxAddToQuery` are called to update `iqsState.iqsVBState` and `iqsState.iqsQuery`, respectively.

Batch Mode behaviour:

Except that updating the set of previous chosen items, cleaning the current lists, checking if `attrname` was the last chosen (class)attribute or facet and calling `iqsSAauxSetVBState` do not occur, the Assisted Mode description applies to the Batch Mode.

Return values (`rIQS`):

- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_BATCHNOAOIDGENVAL` - no objects found with the generic attribute `attrname` having the value `attrvalue`; operation aborted; only in Batch Mode;
- `IQS_BATCHNOAOIDFACVAL` - no objects found with the facet `attrname` having the value `attrvalue`; operation aborted; only in Batch Mode;

- `IQS_BATCHNOAOIDATTVAL` - no objects found with the class attribute `attrname` having the value `attrvalue`; operation aborted; only in Batch Mode;
- `IQS_SUCCESS` - operation successful.

`void iqsSAgetAoidsBySlc (char FAR *slc)`

Assisted Mode behaviour:

This function is *Template2* specific. It is invoked after having chosen a software life cycle from an appropriate interface list pane (whose contents are preserved in `iqsState.iqsSLCNames`). `iqsSAgetAoidsBySlc` will invoke `iqsGetAoidsBySLC` in order to filter the query solution, presently in `iqsState.iqsQS`, leaving only the objects associated with the software life cycle given by `slc`. Once filtering of `iqsState.iqsQS` has successfully occurred, the refinement by software life cycle and phases is considered finished, becoming inaccessible, and the lists of available generic attributes, facets and class attributes are cleaned, in order to enforce their update if, later, one decides to make a refinement of that kind. Also, `iqsSAauxAddToQuery` and `iqsSAauxSetVBState` are both called to properly update `iqsState.iqsQuery` and `iqsState.iqsVBState`.

Batch Mode behaviour:

If none of the objects in `iqsState.iqsQS` is associated with `slc` or with any other software life cycle, `iqsSAgetAoidsBySlc` will immediately return, ending the batch solving of the present query. Except that cleaning the current lists of choosable items and calling `iqsSAauxSetVBState` aren't both performed, the Assisted Mode description applies to the Batch Mode.

Return values (`rIQS`):

- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_BATCHNOSLC` - no objects found associated with the software life `slc`; operation aborted; only in Batch Mode;
- `IQS_SUCCESS` - operation successful;

`void iqsSAgetSlcsAndAoidsByPha (char FAR *pha)`

Assisted Mode behaviour:

This function is *Template2* specific. It is invoked after having chosen a software life cycle phase from an appropriate interface list pane (whose contents are preserved in `iqsState.iqsPHANames`). `iqsSAgetSlcsAndAoidsByPha` will invoke `iqsGetSLCsByPHA` in order to know every software life cycles containing the phase pha. Then, for each one of these software life cycles, iqsGetAoidsBySLC is called so that the current query solution (presently in iqsState.iqsQS) is filtered, keeping only the objects associated with the software life cycles actually containing pha. If there was a unique software life cycle containing the provided phase (pha), then both the refinements by phases and software life cycles are considered finished, becoming inaccessible. Otherwise, the phase pha is added to the set iqsState.iqsPHANamesPrevious, so that no longer it will be possible to specialize the query by that phase. Also, the lists of available generic attributes, facets, class attributes, software life cycles and phases are cleaned, to enforce their update in case later one decides to make a refinement (if possible) of that kind. iqsSAauxAddToQuery and iqsSAauxSetVBState are both called to properly update iqsState.iqsQuery and iqsState.iqsVBState.

Batch Mode behaviour:

If none of the objects in iqsState.iqsQS is associated with the phase pha or with a known software life cycle, iqsSAgetSlcsAndAoidsByPha will immediately return, ending the batch solving of the present query. Except that cleaning the current lists of choosable items and calling iqsSAauxSetVBState aren't both performed, the Assisted Mode description applies to the Batch Mode.

Return values (rIQS):

- IQS_PARAMERR -bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_BATCHNOPHA - no objects found associated with the software life phase pha; operation aborted; only in Batch Mode;
- IQS_SUCCESS - operation successful;

```
void iqsSAafterIsCompoundPressed()
```

Assisted Mode behaviour:

This function is called only in the case of a *Template3* query variant, immediately after the IsCompound button has been pressed, making the AND IS COMPOUND tokens to be joined to the query phrase. The flag iqsState.IsCompoundPressed is then enabled and iqsState.iqsQS is filtered by iqsGetCompounds. This will leave in iqsState.iqsQS only the compound objects from the primitive iqsState.iqsQS content. Once this filtering has been successfully done, the lists of available generic attributes, facets and class attributes are cleaned, in order to enforce their update if, later, one decides to make a refinement of the same kind (this time over compound objects). Finally iqsSAauxSetVBState and iqsSAauxAddToQuery are called to update iqsState.iqsVBState and iqsState.iqsQuery, respectively.

Batch Mode behaviour:

If none of the objects in iqsState.iqsQS is compound, iqsSAafterIsCompoundPressed immediately returns, ending the batch solving of the present query. The flag iqsState.IsCompoundPressed is left unchanged, cleaning the current lists of choosable items doesn't take place and iqsSAauxSetVBState is not called. The remain of the Assisted Mode description applies to the Batch Mode.

Return values (rIQS):

- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_NOTFOUND - no compounds found; only in Assisted Mode;
- IQS_BATCHISCOMPOUNDNOCLAOS - no compounds were found; operation aborted; only in Batch Mode;
- IQS_SUCCESS - operation successful;

```
void iqsSAgetAoidsByCaractRel(char FAR *crlname)
```

Assisted Mode behaviour:

This function is called during a *Template3* or *Template4* query variant, after choosing, in the appropriate interface list pane, one of the available characteristic relations (received in the crlname parameter). iqsSAgetAoidsByCaractRel will call iqsGetClustersByCaractRel in order to obtain from iqsState.QS (or iqsState.QSC, in the *Template4* query variant) only the objects being clusters and having the crlname characteristic relation. If this is successfully accomplished, crlname is joined to set of previous chosen characteristic relations (iqsState.iqsCRLNamesPrevious), preventing users from choosing later the same characteristic. Also, if crlname was the last item of iqsState.iqsCRLNames, then this refinement path is considered explored, becoming inaccessible, and the lists of available generic attributes, facets and class attributes are cleaned, in order to enforce their update if, later, one decides to make a refinement of that kind. Finally iqsSAauxSetVBState and iqsSAauxAddToQuery are called to update iqsState.iqsVBState and iqsState.iqsQuery, respectively.

Batch Mode behaviour:

Except that updating, the set of previously chosen characteristic relations, cleaning the current lists, checking if crlname was the last chosen characteristic relation and calling iqsSAauxSetVBState do not occur, the Assisted Mode description applies to the Batch Mode.

Return values (rIQS):

- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_BATCHNOCRLS - no clusters found with the characteristic relation crl;
- IQS_SUCCESS - operation successful;

```
void iqsSAafterBelongToCompoundPressed()
```

Assisted Mode behaviour:

This function is called only in the case of a *Template4* query variant and immediately after the `BelongToCompound` button has been pressed, making the `AND BELONG TO COMPOUND` tokens to be added to the query phrase. The flag `iqsState.BelongToCompoundPressed` is enabled and a copy of `iqsState.iqsQS` is made to `iqsState.iqsQSC` so that `iqsGetClaoByMember` is invoked over it, leaving there the compound objects containing the ones presently in `iqsState.iqsQS`. Once this has been done, the lists of available generic attributes, facets and class attributes are cleaned, in order to enforce their update if, later, one decides to make a refinement of the same kind (this time, however, over the compound objects kept by `iqsState.iqsQSC`). Finally `iqsSAauxAddToQuery` and `iqsSAauxSetVBState` are called to update `iqsState.iqsQuery` and `iqsState.iqsVBState`, accordingly.

Batch Mode behaviour:

If none of the objects in `iqsState.iqsQS` is member of a compound, `iqsSAafterBelongToCompoundPressed` immediately returns, ending the batch solving of the present query. The flag `iqsState.BelongToCompoundPressed` is left unchanged, cleaning the current lists of choosable items doesn't take place and `iqsSAauxSetVBState` is not called. The remain of the Assisted Mode description applies to the Batch Mode.

Return values (`rIQS`):

- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or uknown error; operation aborted;
- `IQS_NOTFOUND` - no compounds found containing the present query solution; only in Assisted Mode;
- `IQS_BATCHBELONGTOCOMPOUNDNOCLAOS` - no compounds were found containing the present query solution; operation aborted; only in Batch Mode;
- `IQS_SUCCESS` - operation successful.

```
void iqsSAinitQuery(int template)
```

Assisted Mode behaviour:

This function is called every time one of the interface buttons `#5` to `#8` is pressed (making one of the tokens of the format `TEMPLATEx`[32] to be added to the query text). `iqsSAinitQuery` will start the internal query phrase synthesis with the text `"#index TEMPLATEx "`, where `index` is returned by `iqsGetNextGlobalHIndex`, and `x` (depending on the `template` parameter),

[32]x varying from 5 to 8.

will assume a value between 5 and 8; `iqsSAauxSetVBState` will set the next interface state.

Batch Mode behaviour:
Except that `iqsSAauxSetVBState` is not called, the rest of the function behaves like in Assisted Mode.

Return values (`rIQS`):
• `IQS_NOMEMORY` - not enough memory; operation aborted;
• `IQS_SUCCESS` - operation successful.

`void iqsSAgetAoidsFromQuery(int query)`

Assisted Mode behaviour:
This function is called during a non-kernel query synthesis (*Template5* to *Template8* variants). It starts by puting into `iqsState.iqsQS` the objects of a previously solved query, whose History index is given by the `query` parameter (after validated and converted from a local to a global reference on the History). In the case of a *Template5* and *Template6* query variants, `iqsSAgetAoidsFromQuery` will filter `iqsState.iqsQS`, by calling `iqsGetSourcesAndLinks` and `iqsGetSources`, respectively, so that only source objects will remain there. Additionally, the outgoing links are also retrieved in a *Template5* query, allowing for the `iqsState.iqsLNKNames` set to be initialized for later use. `iqsSAauxAddToQuery` and `iqsSAauxSetVBState` are also called to update `iqsState.iqsQuery` and `iqsState.iqsVBState`, accordingly.

Batch Mode behaviour:
Except that `iqsSAauxSetVBState` is not called, the rest of the function behaves as in Assisted Mode.

Return values (`rIQS`):
• `IQS_PARAMERR` - bad parameters; operation aborted;
• `IQS_NOMEMORY` - not enough memory; operation aborted;
• `IQS_ERROR` - internal or unknown error; operation aborted;
• `IQS_NOSOURCES` - no sources found; only in Assisted Mode;
• `IQS_BATCHINDEXNOTVALID` - unexpected index; operation aborted; only in Batch Mode;
• `IQS_BATCHNOSOURCES` - no sources found; only in Batch Mode;
• `IQS_SUCCESS` - operation successful;

`void iqsSAgetSourcesByLink(char FAR * link)`

Assisted Mode behaviour:
This function is called both in *Template5* and *Template6* variants, after choosing a `link` from an interface list pane, whose contents are preserved by

iqsState.iqsLNKNames. It intends to leave in iqsState.iqsQS only the objects being sources of the specified link. Therefore, for each source in iqsState.iqsQS, the list of his links is searched for the presence of link (the set of these lists of links is kept by iqsState.iqsListList[1], initialized during iqsSAgetAoidsFromQuery for the *Template5* queries and initialized during iqsSAgetSourcesAndLinksBySinks for the *Template6* queries). If the link is found there, then the current object is assumed to be a source for that link. iqsSAauxAddToQuery and iqsSAauxSetVBState are also called to update iqsState.iqsQuery and iqsState.iqsVBState, accordingly.

Batch Mode behaviour:
Except that iqsSAauxSetVBState is not called, the rest of the function acts as in Assisted Mode.

Return values (rIQS):
- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_BATCHNOLINK - no source found for link; operation aborted; only in Batch Mode;
- IQS_SUCCESS - operation successful.

```
void iqsSAgetSourcesByLinkAndSinks(char FAR * link,
                                   int query)
```

This function is *Template5* specific and it is invoked after choosing (for the second time during the query synthesis), a History query, of index query (which is validated and converted from a local to a global reference). The objects associated with this previously solved query are submited, with iqsState.iqsQS and link, to iqsGetSourcesByLinkAndSinks, so that every object in iqsState.iqsQS is checked to see if it is a source of the relation link, to at least one sink in the set of objects of the History query. Only the sources obtained that way will remain in iqsState.iqsQS. iqsSAauxAddToQuery and iqsSAauxSetVBState are also called to update iqsState.iqsQuery and iqsState.iqsVBState, accordingly.

Batch Mode behaviour:
Except that iqsSAauxSetVBState is not called, the rest of the function acts as in Assisted Mode.

Return values (rIQS):
- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_NOSINKS - no sinks were found in the History query for the link outgoing from iqsState.iqsQS; operation aborted; only in Assisted Mode;

- IQS_BATCHNOSINKS - no sinks were found in the History query for the link outgoing from iqsState.iqsQS; operation aborted; only in Batch Mode;
- IQS_BATCHINDEXNOTVALID - unexpected index; operation aborted; only in Batch Mode;
- IQS_SUCCESS - operation successful;

---

void iqsSAgetSourcesAndLinksBySinks(int query)

iqsSAgetSourcesAndLinksBySinks is *Template6* specific and it is invoked after choosing a History query (for the second time during the query synthesis), of index query (which is validated and converted from a local to a global reference). Both the objects currently in iqsState.iqsQS and the ones associated with this History query, plus the iqsState.iqsListList, are submited to iqsGetSourcesAndLinksBySinks, so that in iqsState.iqsQS will only remain the sources of at least one link to at least one sink in the History query (iqsState.iqsListList will have, in turn, a specific list of outgoing links for each source found). If this filtering ends successfully, iqsMakeSet will be called to initialize iqsState.iqsLNKNames based on the contents of iqsState.iqsListList.iqsSAauxAddToQuery and iqsSAauxSetVBState are called to update iqsState.iqsQuery and iqsState.iqsVBState, accordingly.

Batch Mode behaviour:
Except that iqsSAauxSetVBState is not called, the rest of the function acts as in Assisted Mode.

Return values (rIQS):
- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_NOSOURCE - no links found between iqsState.iqsQS and the History query; operation aborted; only in Assisted Mode;
- IQS_BATCHNOSOURCE - no links found between iqsState.iqsQS and the History query; operation aborted; only in Batch Mode;
- IQS_BATCHINDEXNOTVALID - unexpected index; operation aborted; only in Batch Mode;
- IQS_SUCCESS - operation successful;

---

void iqsSAqueryUnion(int query2)

This function is *Template7* specific and it is invoked after choosing a History query (for the second time during the query synthesis), of index query2 (which is validated and converted from a local to a global reference). The objects associated with this query2 are joined, via iqsSetUnion, with the ones presently in `iqsState.iqsQS`. `iqsSAauxAddToQuery` and `iqsSAauxSetVBState` are also called to update `iqsState.iqsQuery` and `iqsState.iqsVBState`, accordingly.

Batch Mode behaviour:
Except that `iqsSAauxSetVBState` is not called, the rest of the function acts as in Assisted Mode.

Return values (`rIQS`):
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_BATCHINDEXNOTVALID` - unexpected index; operation aborted; only in Batch Mode;
- `IQS_SUCCESS` - operation successful.

6.1 The IQS Semantic Actions Auxiliary API

`void iqsSAauxSetVBState(bool b1, ..., bool b25)`

This procedure receives the logical values (enable/disable) that `iqsState.iqsVBState` structure fields must assume in order to reflect the state of the query synthesis. Recall to section **4.2.2** for a brief description of those fields. `iqsSAauxSetVBState` is specific to Assisted Mode.

`int iqsSAauxAddToQuery(char FAR *string)`

This function is responsible for appending the `string` parameter (a set of syntactically and semantically valid tokens), to the query phrase, `iqsState.iqsQuery`, presently under construction.

Return values:
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_SUCCESS` - operation successful.

Secondary effects:
- `IQS_NOMEMORY` - cleans `iqsState.iqsQuery`.

`int iqsSAauxAddQueryToHistory()`

The operation of adding a solved query to the History is performed after a successful query resolution. `iqsSAauxAddQueryToHistory` first adds the query text, presently in iqsState.iqsQuery, and then the respective objects, in iqsState.iqsQS.

Return values:
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_SUCCESS - operation successful.

```
int iqsSAauxInitAttrLists()
```

This function is called by iqsSAafterAttrTypeChoice every time a refinement by generic attributes, facets, class attributes, software life cycles, phases or characteristic relations is initiated by pressing the respective interface button. The goal is to initialize or update the iqsState fields iqsState.iqsAOGAttribs (through iqsGetAOGAttribs), iqsState.iqsFACAttribs (through iqsGetFacets), iqsState.iqsCLAAttribs (through iqsGetClassAttributes), iqsState.iqsSLCNames (through iqsGetSLCs), iqsState.iqsPHANames (through iqsGetPHAs) and iqsState.iqsCRLNames (through iqsGetCaractRel) only with the semantically valid tokens in the context of the current query solution, that is, only the generic attributes, facets, etc, defined by the objects in iqsState.iqsQS will be of interest and become available to the user. Also, in each case, the respective set of previously chosen tokens is removed from the one here obtained, in order to prevent the users to re-enter old (and so redundant) refinement paths. iqsSAauxInitAttrLists is specific to Assisted Mode.

Return values (rIQS):
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOTFOUND - no generic attributes, facets, class attributes, software life cycles, phases or characteristic relations available;
- IQS_SUCCESS - operation successful.

```
int iqsSAauxSelectAoidsByValue(char FAR *value)
```

iqsSAauxSelectAoidsByValue is called by iqsSAgetAoidsByValue to project iqsState.iqsQSV over iqsState.iqsQS (iqsState.iqsQSC in the case of a *Template4* query variant). That is, only the objects whose values in iqsState.iqsQSV are equal to value shall remain in iqsState.iqsQS (or iqsState.iqsQSC in the case of *Template4*). However, if iqsState.iqsQSV.distance is greater than -1, this means that iqsState.iqsQSV is a set of facets values and so, the conceptual distance field, iqsState.iqsQSV.distance, should be considered during iqsState.iqsQS filtering, that is, only the objects whose facets values in iqsState.iqsQSV have a conceptual distance, from the parameter value, of at least `iqsState.iqsQSV.distance`, will remain in `iqsState.iqsQS` (or `iqsState.iqsQSC` in the case of *Template4*). This special case is handled by calling the `ctsSearchArc` function.

Note that if `value` is an empty string, `iqsState.iqsQSV.distance` will be ignored, and a simple projection takes place.

If `value` is not an empty string and `iqsState.iqsQSV.distance` is greater than `-1`, then `ctsSearchArc` will not be called every time the value retrieved from from `iqsState.iqsQSV.info` is an empty string.

Return values:

- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - unknown `value` (not found in `iqsState.iqsQSV.info`); operation aborted;
- `IQS_SUCCESS` - operation successful.

Secondary effects:

- `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND` - cleans `iqsState.iqsQS` (`iqsState.iqsQSC`);

7 The IQS Visual Basic related API

The next state of the Visual Basic interface layer is kept at `iqsState.iqsVBstate`. Aditionaly, the contents of some list-pannes are also maintained by some specific fields of the `iqsState` global variable[33]. These structures are updated, in a deterministic way, every time a semantic actions is executed. In order to let the Visual Basic interface layer to reflect the contents of these variables, a set of exportable functions, allowing for the retrieval of that information, must be provided. Also, functions to reset the `iqsState` fields when starting (or during or terminating) an IQS session, are needed. Finaly, a way must be provided to invoke the IQL Parser, with the query (or batch of querys) text.

The following functions take care of the previous subjects; they can be found at **iqs.c** file.

```
int WINAPI __export VBiqsResetParser(int mode)
```

Depending on the `mode` parameter, the `VBiqsResetParser` function will initialize (or deallocate) some specific `iqsState` fields, namely some sets of names and values, flags, the `iqsVBstate` structure, etc[37].

[33]recall **4.2 The *ParserState* data type**. The History is handled separately because there are circumstances in which `VBiqsResetParser` is called but the History must be left intact.

`mode` possible values:

- `IQS_START, IQS_STARTBATCH` - immediatly after entering an IQS session or changing the IQS operation mode;
- `IQS_NEXT` - immediatly before a query synthesis (in Aided Mode) or immediatly before processing the next query of a batch (in Batch Mode);
- `IQS_END` - immediatly before leaving an IQS session or changing the IQS operation mode;

Return values:

- `IQS_SUCCESS` - operation (always) successful.

`int WINAPI __export VBiqsInitHistory()`

This function exclusively initializes the `iqsState.iqsQueryHistory` field.

Return values:

- `IQS_SUCCESS` - operation (always) successful.

`int WINAPI __export VBiqsClearHistory()`

`VBiqsClearHistory` will call `iqsCleanSet` in order to deallocate and reinitialize the History structure, `iqsState.iqsQueryHistory`. `iqsSAauxSetVBState` is also called so that the Template buttons (in the Assisted Mode) and the History and Display menus (in both operation Modes) reflect the empty state of the History structure.

Return values:

- `IQS_SUCCESS` - operation (always) successful.

`int WINAPI __export VBiqsSaveHistory(LPSTR VBfile)`

This function will save, at the file specified by `VBfile`, the text of the querys kept by the History.

Return values:

- `IQS_BATCHIOERROR` - open error or write error over `VBfile` file;
- `IQS_SUCCESS` - operation successful.

`int WINAPI __export VBiqsGetVBState(VBState *state)`

This function will make a copy, field by field, of the `iqsState.iqsVBState` structure into the `state` parameter. The `state` parameter should be a pointer to a Visual Basic structure of the type `VBiqsState`. `VBiqsGetVBState` allows the information concerning the enable/disable state of the interface buttons and list panes to access to the Visual Basic layer.

Return values:
- `IQS_SUCCESS` - operation (always) successful.

```
int WINAPI __export VBiqsGetQuery(HLSTR query)
```

`VBiqsGetQuery` will call `VBSetHlstr` so that a copy of `iqsState.iqsQuery` is made to `query`. This is how a copy of the query, as synthesised by the semantic actions, can access the Visual Basic layer.

Return values:
- `IQS_SUCCESS` - operation (always) successful.

```
int WINAPI __export VBiqsGetNameFromNameList(
                HLSTR VBname, int namelist, int mode)
```

This function will return, on successive calls (first call with `mode`=`IQS_START` and the following ones with `mode`=`IQS_NEXT`), all the names contained in the set of names coded in the `namelist` integer parameter. `VBiqsGetNameFromNameList` will call `VBSetHlstr` to make `VBname` a copy of the present name of the list of names being scanned. By invoking `VBiqsGetNameFromNameList` until receiving `IQS_NOTFOUND`, the Visual Basic layer expects to receive the contents, one by one, of a list of names (most of the times, to be displayed at an interface list pane). The parameter `namelist` is checked only if `mode`=`IQS_START`.

`namelist` possible values:
- `IQS_AOGNAMELIST` - `iqsState.iqsAOGAttribs` will be scanned;
- `IQS_FACNAMELIST` - `iqsState.iqsFACAttribs` will be scanned;
- `IQS_CLANAMELIST` - `iqsState.iqsCLAAttribs` will be scanned;
- `IQS_SLCNAMELIST` - `iqsState.iqsSLCNames` will be scanned;
- `IQS_PHANAMELIST` - `iqsState.iqsPHANames` will be scanned;
- `IQS_CRLNAMELIST` - `iqsState.iqsCRLNames` will be scanned;
- `IQS_LNKNAMELIST` - `iqsState.iqsLNKNames` will be scanned.

`mode` possible values:
- `IQS_START` - get the first name;
- `IQS_NEXT` - get the next name.

Return values:
- IQS_NOTFOUND - no (more) names available at the list of names specified by namelist;
- IQS_SUCCESS - operation successful.

```
int WINAPI __export VBiqsGetValuesFromQSV(
                             HLSTR VBvalue, int mode)
```

VBiqsGetValuesFromQSV will return, on successive calls (first call with mode=IQS_START and the following ones with mode=IQS_NEXT), all the values contained in the iqsState.iqsQSV. VBiqsGetValuesFromQSV will call VBSetHlstr to make VBvalue a copy of the present value retrieved from the iqsState.iqsQSV list. By invoking VBiqsGetValuesFromQSV until receiving IQS_NOTFOUND, the Visual Basic layer expects to receive the contents of iqsState.iqsQSV, one by one and without redundant or empty values.

mode possible values:
- IQS_START - get the first iqsState.iqsQSV value;
- IQS_NEXT - get the next iqsState.iqsQSV value.

Return values:
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_NOTFOUND - no (more) values available at iqsState.iqsQSV;
- IQS_SUCCESS - operation successful.

```
int WINAPI __export VBiqsGetHistory(HLSTR VBquery,
                                    int mode)
```

This function will return, on successive calls (first call with mode=IQS_START and the following ones with mode=IQS_NEXT), the text of all the queries contained at the History. VBiqsGetHistory will call VBSetHlstr to make VBquery a copy of the present query text retrieved from iqsState.iqsQueryHistory.

mode possible values:
- IQS_START - get the text of the first query kept by the History (iqsState.iqsQueryHistory.querys[0].querytext);
- IQS_NEXT - get the text of the next query of the History.

Return values:
- IQS_NOTFOUND - no (more) queries available at iqsState.iqsQueryHistory;
- IQS_SUCCESS - operation successful.

```
int WINAPI __export VBiqsGetAoidsFromHistory(
                                          int query);
```

`VBiqsGetAoidsFromHistory` will make `iqsState.iqsQS` a copy of the AOIDs of the `query`th query of the History. That copy will be mostly used by some Result Manager facilities.

Return values:
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_INDEXNOTVALID` - invalid index; operation aborted
- `IQS_SUCCESS` - operation successful.

```
int WINAPI __export VBiqsGetResult(
                        LPAOENTRYGEN aoGen,int mode)
```

This function will return, on successive calls (first call with `mode=IQS_START` and the following ones with `mode=IQS_NEXT`), some of the genneric information (via `aoGen`) of every AOID of `iqsState.iqsQS`. This data will be depicted in a table, just after having pressed the `CHECK` button (during the query synthesis), in Aided Mode.

`mode` possible values:
- `IQS_START` - gets the generic data for the first AOID in `iqsState.iqsQS;`
- `IQS_NEXT` - gets the generic data for the next AOID in `iqsState.iqsQS;`

Return values:
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no (more) objects available at `iqsState.iqsQS`;
- `IQS_SUCCESS` - operation successful.

```
int WINAPI __export VBiqsShowFAC(void)
int WINAPI __export VBiqsShowLNK(void)
int WINAPI __export VBiqsShowMBR(void)
```

These three functions are in charged of initializing the Result Manager appropriate data structures in order to display facets, links or members related information, concerning the objects kept in `iqsState.iqsRM`. These objects are a copy `of iqsState.iqsQS` (or `iqsState.iqsQSC`), which has the temporary or final solution of the query being made, or from a specific query of the History.

Return values:
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_SUCCESS` - operation successful.

```
int WINAPI __export VBiqsParser(LPSTR VBstring)
```

In <u>Assisted Mode</u>, the Visual Basic layer will call `VBiqsParser` whenever it wants the query phrase `VBstring` to be recognized and solved. `VBiqsParser` will, in turn, call `yyparse` with a local copy of `VBstring`.

Return values:
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_PARSERROR` - parser internal error; operation aborted;
- every other possible integer code returnable in Assisted Mode

---

```
int WINAPI __export VBiqsBatchParser(LPSTR VBfile)
```

In <u>Batch Mode</u>, the Visual Basic layer will call `VBiqsBatchParser` whenever it wants a batch of queries to be solved. This set of query phrases is presumed to be kept in a file whose path name is given by the `VBfile` parameter. After the file has been loaded into memory, and `iqsSetNextLocalHIndex` has been called with `0` (zero) (in order to start a new local context of references on History), `VBiqsBatchParser` will invoke `yyparse` with the batch (in memory) to be solved. During this process, every time a query is successfully terminated, `iqsSAcheck` is called in order to refresh a log file, `logfile.iqs`, with the contents of `iqsState.iqsQS` and the suffix `SUCCESS`. However, if an error occurs during the parsing process or inside a semantic action, the remaining queries of the batch are ignored (not solved). In this case, the log file will contain only the text of the queries successfully solved (and the respective suffix, `SUCCESS`) as well as the part of the query text managed to be solved just before the error took place (this late text and a suffix indicating the error is provided by `VBiqsBatchParser` as soon as `yyparse` returns).

Return values (appearing also as suffix tokens):
- `IQS_PARSERROR` - parser internal error; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_BATCHIOERROR` - I/O error over `logfile.iqs` or `VBfile`; operation aborted;
- `IQS_BATCHNOMEMORY`- not enough memory to load `VBfile`; operation aborted;
- `IQS_BATCHNOOBJS` - no objects available for the specified class; operation aborted;
- `IQS_BATCHNOGENATTR` - unknown generic attribute; operation aborted;
- `IQS_BATCHNOFACATTR` - unknown facet; operation aborted;
- `IQS_BATCHNOATTATTR` - unknown class attribute; operation aborted;
- `IQS_BATCHNOGENVAL` - no values found for the specified generic attribute; operation aborted;

- `IQS_BATCHNOFACVAL` - no values found for the specified facet; operation aborted;
- `IQS_BATCHNOATTVAL` - no values found for the specified class attribute; operation aborted;
- `IQS_BATCHNOAOIDGENVAL` - no objects found with the specified generic attribute; operation aborted;
- `IQS_BATCHNOAOIDFACVAL` - no objects found with the specified facet; operation aborted;
- `IQS_BATCHNOAOIDATTVAL` - no objects found with the specified class attribute; operation aborted;
- `IQS_BATCHNOSLC - n`o objects found with the specified software life cycle; operation aborted;
- `IQS_BATCHNOPHA - n`o objects found with the specified software life cycle phase; operation aborted;
- `IQS_BATCHISCOMPOUNDNOCLAOS` - no compounds available in the previous selected objects; operation aborted;
- `IQS_BATCHNOCRLS` - no objects found with the specified characteristic relation; operation aborted;
- `IQS_BATCHBELONGTOCOMPOUNDNOCLAOS` - no compounds found containing the previous selected objects; operation aborted;
- `IQS_BATCHINDEXNOTVALID` - invalid History index; operation aborted;
- `IQS_BATCHNOSOURCES` - no sources available in the previous selected objects; operation aborted;
- `IQS_BATCHNOLINK` - no source available for the specified link, in the previous selected object; operation aborted;
- `IQS_BATCHNOSINKS` - no sinks available for the specified link, in the last selected objects; operation aborted;
- `IQS_BATCHNOSOURCE` - no source for the specified sinks, in the previous selected sources; operation aborted;
- `IQS_SUCCESS` - operation successful.

---

8 IQS module cross reference

This chapter shows the global cross reference for all of the functions of the IQS module. The IQS module functions make internal calls[34] as well as external calls to the ERA, CON, RM and CTS modules. Thus, the provided description will be based on a field for the name of the caller IQS function and, whenever necessary, specific fields for the called functions on other modules.

This cross reference distinguishes between functions implemented in the **iqs.c** and **actions.c** files. For each file, the functions are gathered in groups reflecting their main functionalities.

8.1 Cross reference for the iqs.c file

[34]that is, they also call functions of their own module

8.1.1 IQS API Auxiliary functions

IQS function	IQS calls
iqsFrealloc	
iqsStrtok	
iqsCheckWordInList	iqsStrtok
iqsGetAttrValue	iqsCheckWordInList

8.1.2 Set API functions

IQS function	IQS calls
iqsCleanSet	iqsCleanSet
iqsCopySet	iqsCleanSet
iqsSetDifference	iqsCopySet
	iqsCleanSet
	iqsCheckWordInList
iqsSetIntersection	iqsCopySet
	iqsSetDifference
	iqsCleanSet
iqsSetUnion	iqsCopySet
	iqsCleanSet
	iqsFrealloc
	iqsMakeSet
iqsMakeSet	iqsCleanSet
	iqsFrealloc
	iqsCheckWordInList
	iqsMakeSet
	iqsCopySet

8.1.3 IQS API functions

IQS function	IQS calls	ERA calls	CON calls
iqsGetHierarchyAoids	iqsCleanSet	eraGetObj	
	iqsFrealloc		
	iqsGetAttrValue		
	iqsCopySet		
	iqsSetDifference		
iqsGetAOGAttribs	iqsCleanSet	eraGetObj	
	iqsCheckWordInList		
	iqsSetDifference		
	iqsGetAttrValue		
	iqsFrealloc		
iqsGetAOGValues	iqsCleanSet	eraGetObj	
	iqsGetAttrValue		
	iqsFrealloc		
iqsGetFacets	iqsCleanSet	eraGetObj	
	iqsFrealloc		
	iqsCheckWordInList		
	iqsGetAttrValue		
	iqsSetDifference		
iqsGetFacetsValues	iqsCleanSet	eraGetObj	

	iqsGetAttrValue		
	iqsFrealloc		
iqsGetClassAttributes	iqsCleanSet	eraGetObj	
	iqsGetAttrValue		
	iqsCheckWordInList		
	iqsSetUnion		
iqsGetAttribsValues	iqsCleanSet	eraGetObj	
	iqsGetAttrValue		
	iqsFrealloc		
iqsGetSLCs	iqsCleanSet	eraGetObj	
	iqsGetAttrValue		
	iqsCopySet		
	iqsMakeSet		
iqsGetPHAs	iqsCleanSet		
	iqsGetSLCs		
	iqsCheckWordInList		
	iqsGetPHAsBySLC		
	iqsSetUnion		
	iqsMakeSet		
iqsGetPHAsBySLC	iqsCleanSet		conGetSLCPHA
	iqsFrealloc		
iqsGetAoidsBySLC	iqsCleanSet	eraGetObj	
	iqsGetAttrValue		
	iqsCopySet		
iqsGetSLCsByPHA	iqsCleanSet		
	iqsCheckWordInList		
	iqsGetPHAsBySLC		
	iqsCopySet		
iqsGetCompounds	iqsCleanSet		conGetMbr
	iqsCopySet		

Note: `iqsGetHierarchyAoids` calls also `VCsrvGetClasses` at the SRV module.

iqsGetCaractRel	iqsCleanSet		conGetMbrLnk
	iqsFrealloc		conGetLnk
	iqsSetUnion		
iqsGetClustersByCaractRel	iqsCleanSet		conGetMbrLnk
	iqsFrealloc		conGetLnk
	iqsSetUnion		
	iqsCopySet		
iqsGetClaoByMember	iqsCleanSet		conGetMbr
	iqsFrealloc		
	iqsMakeSet		
iqsGetMemberByClao	iqsCleanSet		conGetMbr
	iqsFrealloc		
	iqsMakeSet		
iqsGetSources	iqsCleanSet		conGetLnk
	iqsCopySet		
iqsGetSourcesAndLinks	iqsCleanSet		conGetLnk
	iqsFrealloct		
	iqsCopySet		
iqsGetSourcesByLinkAndSinks	iqsCleanSet		conGetLnk
	iqsCopySet		
iqsGetSourcesAndLinksBySinks	iqsCleanSet		conGetLnk
	iqsFrealloc		

	iqsCopySet		

8.1.4 IQS Visual Basic related API functions

IQS function	IQS calls	RM calls
VBiqsResetParser	iqsSAauxSetVBState	
	iqsCleanSet	
VBiqsInitHistory		
VBiqsClearHistory	iqsCleanSet	
	iqsSAauxSetVBState	
VBiqsSaveHistory		
VbiqsGetVBState		
VbiqsGetQuery		
VbiqsGetNameFromNameList	iqsCheckWordInList	
VbiqsGetValuesFromQSV	iqsMakeSet	
	iqsSetDifference	
	iqsCleanSet	
	iqsCheckWordInList	
VBiqsGetHistory		
VBiqsGetAoidsFromHistory	iqsCopySet	
VBiqsGetResult		
VBiqsShowFAC	iqsCopySet	rmReset
	iqsGetMemberByClao	rmAddVertex
	iqsSetIntersection	rmAddArc
VBiqsShowLNK	iqsCopySet	rmReset
	iqsGetMemberByClao	rmAddVertex
	iqsSetIntersection	rmAddArc
VBiqsShowMBR	iqsCopySet	rmReset
	iqsGetMemberByClao	rmAddVertex
	iqsSetIntersection	rmAddArc
VBiqsParser		
VBiqsBatchParser	iqsSAauxSetVBState	

8.2 Cross reference for the actions.c file

8.2.1 IQS Semantic Actions Auxiliary API functions

IQS function	IQS calls	CTS calls
iqsSAauxSetVBState		
iqsSAauxAddToQuery	iqsFrealloc	
iqsSAauxAddQueryToHistory	iqsFrealloc	
	iqsCopySet	
iqsSAauxInitAttrLists	iqsGetAOGAttribs	
	iqsSetDifference	
	iqsGetFacets	
	iqsGetClassAttributes	
	iqsMakeSet	
	iqsGetSLCs	
	iqsGetPHAs	
	iqsGetCaractRel	
iqsSAauxSelectAoidsByValue	iqsCheckWordInList	ctsSearchArc

	iqsSetUnion	
	iqsCleanSet	
	iqsCopySet	

8.2.2 IQS Semantic Actions API functions

IQS function	IQS calls
iqsSAcheck	iqsSAauxAddQueryToHistory
	iqsGetMemberByClao
	iqsSetIntersection
	VBiqsResetParser
iqsSAabort	VBiqsResetParser
batchIqsSAcheckIndex	
iqsSAinitGetAllClass	iqsSAauxAddToQuery
	iqsSAauxSetVBState
iqsSAgetAoidsBellowClass	iqsGetHierarchyAoids
	iqsSAauxInitAttrLists
	iqsSAauxAddToQuery
	iqsSAauxSetVBState
iqsSAafterAttrTypeChoice	iqsSAauxInitAttrLists
	iqsSAauxSetVBState
iqsSAgetAttrValues	iqsCleanSet
	iqsGetAOGValues
	iqsGetFacetsValues
	iqsGetAttribsValues
	iqsSAauxSetVBState
iqsSAgetAoidsByValue	iqsSAauxSelectAoidsByValue
	iqsSetUnion
	iqsSetDifference
	iqsCleanSet
	iqsSAauxSetVBState
	iqsSAauxAddToQuery
iqsSAgetAoidsBySlc	iqsGetAoidsBySLC
	iqsSAauxAddToQuery
	iqsCleanSet
	iqsSAauxSetVBState
iqsSAgetSlcsAndAoidsByPha	iqsGetSLCs
	iqsGetSLCsByPHA
	iqsCheckWordInList
	iqsClopySet
	iqsGetAoidsBySLC
	iqsSetUnion
	iqsMakeSet
	iqsSAauxAddToQuery
	iqsSAauxSetVBState
	iqsCleanSet
iqsSAafterIsCompoundPressed	iqsCopySet
	iqsGetCompounds
	iqsSAauxSetVBState
	iqsCleanSet
	iqsSAauxAddToQuery
iqsSAgetAoidsByCaractRel	iqsGetClustersByCaractRel
	iqsSetUnion

	iqsSetDifference
	iqsCleanSet
	iqsSAauxSetVBState
	iqsSAauxAddToQuery
iqsSAafterBelongToCompoundPressed	iqsCopySet
	iqsGetClaoByMember
	iqsCleanSet
	iqsSAauxSetVBState
	iqsSAauxAddToQuery
iqsSAinitQuery	iqsSAauxAddToQuery
	iqsSAauxSetVBState
iqsSAgetAoidsFromQuery	iqsCopySet
	iqsGetSourcesAndLinks
	iqsMakeSet
	iqsSAauxSetVBState
	iqsGetSources
	iqsSAauxAddToQuery
iqsSAgetSourcesByLink	iqsCheckWordInList
	iqsSetUnion
	iqsCleanSet
	iqsCopySet
	iqsSAauxSetVBState
	iqsSAauxAddToQuery
iqsSAgetSourcesByLinkAndSinks	iqsCopySet
	iqsGetSourcesByLinkAndSinks
	iqsCleanSet
	iqsSAauxSetVBState
	iqsSAauxAddToQuery
iqsSAgetSourcesAndLinksBySinks	iqsCopySet
	iqsGetSourcesAndLinksBySinks
	iqsCleanSet
	iqsMakeSet
	iqsSAauxSetVBState
	iqsSAauxAddToQuery
iqsSAqueryUnion	iqsSetUnion
	iqsSAauxAddToQuery
	iqsSAauxSetVBState

References

[IQS-2.1] "Intelligent Query System - Functional Specification & Architecture", Version: 2, Revision: 1.

[GF93] "Como redireccionar o *input* de um reconhecedor baseado em *lex* e *yacc* para uma zona de memória", Geraldina Fernandes - Universidade do Minho,1993

[CM-1.4 1993] *Comparator & Modifier*. Functional Specification & Architecture. Version: 1; Revision: 4; Workpackage WP2B of Collaboration Offer by INESC.