The OBSCURE Manual

Part I: Editing and Rapid Prototyping

A 03 / 91

Version of February 1991

AUTHORS: Jürgen Fuchs, Annette Hoffmann, Liane Meiss, Joachim Philippi, Michael Stolz, Markus Wolf, Jörg Zeyer

EDITED BY Jacques Loeckx and Markus Wolf

This is the first version of a manual for an implementation of the specification language **OBSCURE**[1]. It without doubt still contains errors and some sections may be unclear or even obscure. The editors are grateful for any comment or suggestion.

Jacques Loeckx, Markus Wolf
Fachbereich 14 (Informatik)
Universität des Saarlandes
D-6600 Saarbrücken
e-mail: loeckx@cs.uni-sb.de
lupus@cs.uni-sb.de

[1]The development of **OBSCURE** has been supported by the Deutsche Forschungsgemeinschaft,

How to use the present manual

The present manual may be used as a tutorial, as a reference manual or as a description of the OBSCURE system. Before providing some hints to the reader we shortly indicate the contents of the different chapters.

The first chapter presents a short introduction to the system components and the *Emacs* editor which is used as a user environment for the system. The notational conventions adhered to throughout the manual are indicated.

The second chapter constitutes a protocol of a session with the system.

The third chapter presents the syntactical constructs of the specification language OBSCURE and their semantics. For a detailed description of the algorithmic specification method and the specification language OBSCURE the reader is referred to [Lo 87] and [LL 90] respectively.

The fourth chapter describes the OBSCURE system proper. All available commands of the system are shortly discussed.

Examples of OBSCURE specifications and an illustration of the work with some system components are to be found in the fifth chapter.

The final chapter explains how to call the components of the system from a standard UNIX shell.

The appendices give an index of the variables and functions of the system, a complete context free syntax for the specification language, a non trivial example of a specification in OBSCURE, an installation guide and references to the literature.

The beginner using the present manual as a **tutorial** should have a quick look at Section 1.1, Section 4.1 and —if necessary— to Appendix E. He/She should then work through Chapter 2 by repeating the session described on a terminal. He/She may also consult Chapter 5.

When using the manual as a **reference manual** one will be interested in particular in Chapter 4 and the Appendices A and B.

The reader interested in a **description** of the OBSCURE system is suggested to read Section 1.1 and Chapter 3. He/She may then have a look at Chapter 2 and Chapter 5.

Contents

Chapter 1

Introduction

The OBSCURE system is a specification environment embedded in the *Emacs* editor. It supports the design, test and maintenance of large specifications written in a specification language also called OBSCURE.

The components of the OBSCURE system are:

- a parser;
- a set of module databases;
- an interpreter;
- a Source-To-Source-Translator (SoToSoTra);
- a theorem prover.

The theorem prover is still under development. Its structure and use will be described in Part II of the OBSCURE manual.

The *Emacs* editor constitutes the core of the system. All components of the system can be called from *Emacs*, as well as from a standard UNIX-shell.

The following section gives a short overview of the four components of the OBSCURE system.

1.1 An overview of the components of the system

1.1.1 The parser

The parser checks the specification text for correctness with respect to the context free grammar and the context conditions of the specification language OBSCURE (see 3 [The specification Language OBSCURE], page 38 for more information). Successfully parsed specifications are entered into the user's module database by the parser.

1.1.2 The module database

There are two types of module databases, a global database **mdbpool** to which all users have reading access, and the personal databases to which the users have reading access and —via the parser— writing access. Each database contains modules, i.e. specifications provided with names. Only the parser can write into a module database. This guarantees that databases contain only syntactically correct specification modules. The user can direct queries to a database through a query language.

1.1.3 The interpreter

The interpreter provides means for *rapid prototyping*. More precisely, the interpreter evaluates terms. These terms have to be terms over the signature of a "closed" specification, i.e. a specification that imports "basic" sorts and operations only (see C [The context free syntax for the specification language], page 96 for more information). The term may contain variables. To this end the interpreter contains a mechanism allowing to assign values to variables. It moreover contains a tracer for debugging and online help.

1.1.4 The Source-To-Source-Translator

The Source-To-Source-Translator (So-To-So-Tra) is a program for translating syntactically correct specifications into a programming language. As for the interpreter a specification to be translated has to be "closed". Currently a translation is possible into C only, but translations into C++ and ML are under development. Note that a specification translated by So-To-So-Tra allows *rapid prototyping* which is in general substantially more efficient than the rapid prototyping performed by the interpreter.

1.1.5 The theorem prover

The theorem prover

- generates formulas the validity of which guarantees the consistency of the specifications (see [LL 90] for more details);
- proves semi-automatically or automatically the validity of these formulas.

The theorem prover is still under development and is to be described in Part II of this Manual.

1.2 On Emacs

OBSCURE is embedded in the *Emacs* editor extended by a special mode of *Emacs*, called **`obscure-mode`**. The syntax of the commands specific to OBSCURE are compatible with the normal command syntax of *Emacs* in order to help the experienced user.

The usual help-functions (see 1.4 [How to get help from ...], page 4), e.g. **`command-apropos`**, **`describe-bindings`**, **`describe-function`** etc., are also available in the **`obscure-mode`**.

The version of *Emacs* used here is that of [Sta 85]. The *Emacs* manual should be available online on your system. There is also a tutorial available on any *Emacs* system. A short introduction to the conventions and the philosophy of *Emacs* is given in the following.

Emacs is an *extensible, customizable, self-documenting, real-time display editor*. In comparison with other text editors it offers additional comfort, such as the simultaneous editing of several files, automatic program indentation and powerful editing commands. Through few keystrokes *Emacs* provides help with concepts, commands, variables, key bindings etc. It is relatively easy to extend the editor by new commands and programs, and it is possible to start other processes from inside the editor. For these reasons *Emacs* was chosen the user environment for OBSCURE.

In *Emacs*, the screen can be divided into several *windows*. At the start of an *Emacs* session only one window is visible. It fills the whole screen except for the last line. The last line, called *echo-area*, echoes entered commands and serves as display area for the so-called *minibuffer*.

A *buffer* in *Emacs* is an '`object`' containing a collection of symbols. Each buffer has a name and a cursor position viz. the position, at which a command takes effect. The contents of a buffer can be made visible in a window. In principle, the number of buffers in one incarnation of *Emacs* is unlimited, but of course only one of them is the active one in which a command takes effect. The user can transform any buffer into the active one by a simple command. Note that there is a file-name associated with each buffer which is not necessarily identical with the name of the buffer.

The *Minibuffer* mentioned above is a special buffer; its contents are visible on the last line of the screen. It is used to read arguments for "complex commands" and to display error messages.

1.3 Notational conventions

As will be described in the following chapters, the commands of the OBSCURE system are called by pressing the ESC or Meta key, then the key labeled x and by entering the command name. Following the notational conventions of the *Emacs* manual [Sta 85] this is denoted by

```
M-x <command name>.
```

Most commands can also be called by a short Control-sequence. This is denoted by

```
C-x
```

and stands for simultaneously pressing the Control-key and the key labeled x. These alternative key sequences are printed as a comment behind the command name as follows

```
M-x <command name>                                          # C-x
```

A short summary of the OBSCURE commands and their key bindings can be found in Appendix A (For more information see A [command index], page 93)

The following notational conventions, concerning variables and arguments to commands are adhered to throughout the present text:

- a variable or argument '`xy`' is written as *xy*;
- arguments that are read interactively after the call of a command are listed in order of their occurrence.

1.4 How to get help from the system

This section presents a few help facilities as offered by *Emacs*. A more complete description of the help facilities can be found in the *Emacs-info-documentation–reader* (see the next but one paragraph for an explanation of the use of the *reader*).

The '`help`'-function is called by the key sequence `C-h`. *Emacs* then asks for an option describing the kind of help wanted. If `C-h` is typed again, a list of all available options together with their meanings is displayed. *Emacs* offers an automatic name completion when asked for help with functions or variables. In case more than one completion is possible, a list of all possible completions is displayed in a separate window that disappears after selecting one of the completions. The completion of the string typed in is started by pressing either `SPC` or `TAB`. While `SPC` only completes up to a symbol different from a letter, `TAB` completes as far as possible.

By typing the option *i* after invoking the help command one starts the *Emacs-info-documentation-reader* which provides an online access to the *Emacs* manual, as well as to the present manual. The '`t`' option starts the *Emacs* Tutorial teaching how to work with *Emacs* interactively.

1.5 Some useful Emacs commands

Note that any command using the minibuffer (e. g. any function call preceded by `M-x`) is saved in a special *history-list*. A few useful commands manipulating the *history-list* are now listed.

`M-x repeat-complex-command` (also called by: `C-x ESC`)

This command restarts any previous command that used the minibuffer. The command asks for the name of the command to be repeated; the default value is the last command executed.

`M-x previous-complex-command` (also called by: `M-p`)

After a **`repeat-complex-command`** the user can wander through the history list or repeat the command just called. This command can be called successively an arbitrary number of times thus allowing to wander through the history list "back in time".

`M-x next-complex-command` (also called by: `M-n`)

This command is the opposite of **`previous-complex-command`** in that it allows to wander "forward in time".

`M-x list-command-history`

The entire history list is displayed in the order of execution. The last command is on the first place. This list is displayed in a newly created buffer called **`*Command History*`**.

Sometimes the user wants to change some of the system variables used by OBSCURE or *Emacs*. This can be achieved by the following command:

`M-x set-variable`

The user is first asked for the name of the system variable to be set. Then he/she is asked for the value of the variable.

Chapter 2

A commented protocol

The aim of this tutorial is to introduce the OBSCURE system by explaining a session with the system. It is advantageous if the reader has an access to the OBSCURE system through a terminal. During the session the three specifications **NAT**, **STACK** and **NATSTACK**, are specified with the help of the system.

The user of the system is supposed to carry the login name `arbor` and it is assumed, that his/her home directory is `/users/arbor` (or `~/` for short). It is further assumed that the user has access to a database called `arbor` which has been created by the command `o-mdb-install`.

The file `/users/arbor/.emacs` should contain the following entry:

```
(load "obscure.elc")
```

For an explanation of this entry see 4.1 [Starting and ...], page 53. If this file does not exist, it should be created before proceeding any further with this tutorial.

The following conventions are used in the text of the now following protocol. Comments are started with the symbol '#'. The symbol '<RET>' stands for typing the key labeled 'RETURN', the symbol `M-x` stands for typing the key with the label 'ESC' or 'META' and then typing the key labeled 'x' (cf. Section 1.3). The symbol `C-x` stands for typing the keys 'CTRL' and 'x' simultaneously (cf. Section 1.3). The symbol <DEL>stands for typing the key labeled 'DEL' or 'Delete'.

The contents of an *Emacs* buffer with the name P–NAME are displayed in the following manner:

```
---------------------- P-NAME ----------------------
|                                                  |
|                                                  |
---------------------- P-NAME ----------------------
```

The contents of the minibuffer (i.e. the last line of the screen) are printed as follows:

MINIBUFFER:

```
<text in the minibuffer>
```

Inputs of the user are preceded by 'INPUT: '. Most commands of *Emacs* can be called either by their name or by a short keysequence. In this tutorial the keysequence is used, but the alternative call is given as a comment. For a detailed description of the commands used see 4 [The OBSCURE System], page 53.

2.1 Starting the system

INPUT: `emacs` # starts *Emacs*

The *Emacs* editor is started and the buffer `*scratch*` is displayed. This buffer is in `fundamental-mode` at the start. If *Emacs* is not available, the *vi* editor can also be used, but the OBSCURE components have to be called from a standard UNIX shell. For an explanation on how to call the components from UNIX see 6 [OBSCURE and UNIX], page 88.

INPUT: `M-o` # call of the function `M-x obscure-mode`

The buffer `*scratch*` is switched to the `obscure-mode`. All key bindings of the OBSCURE system are now available. For more information about the modes of *Emacs* see [Sta 85].

2.2 Editing an atomic specification

The editing of a specification with the OBSCURE system is shown step by step in the following. First, an atomic, i.e. algorithmic specification in the sense of [Lo 87], called STACK , is created. The basic structure of an atomic specification is written into the buffer `STACK` as follows:

INPUT: `C-o a` # call of the function `M-x o-at-spec`

In the minibuffer appears:

MINIBUFFER:

```
buffer name:  (default *scratch*)
```

INPUT: `STACK`<RET>

The empty (new) buffer **STACK** is displayed on the screen and the following question appears in the minibuffer:

MINIBUFFER:

```
Insert specification skeleton ?  (y or n)
```

INPUT: y

The "skeleton" of a specification is generated by the answer 'y'. The new contents of the buffer **STACK** are displayed on the screen:

STACK

```
IMPORTS
 SORTS

 OPNS

CREATE
 SORTS

 OPNS

SEMANTICS
 CONSTRS

 WITH IMPORTED CONSTRS

 VARS

 PROGRAMS

ENDCREATE

E_AXIOMS
 VARS    ;

ENDAXIOMS

I_AXIOMS
 VARS    ;

ENDAXIOMS

FORGET
 SORTS
 OPNS
```

```
## SUBSET OF <s> BY VARS   ; <axiom> ENDSUBSET

## QUOTIENT OF <s> BY VARS   ; <axiom> ENDQUOTIENT
## I_RENAME SORTS <l_sort> AS SORTS <l_sorts>
##      OPNS <l_opn> AS OPNS <l_oname>
## E_RENAME SORTS <l_sort> AS SORTS <l_sorts>
##      OPNS <l_opn> AS OPNS <l_oname>
---------------------------- STACK ----------------------------
```

MINIBUFFER:

```
File to save in:~/
```

INPUT: STACK.T<RET>

The contents of the buffer STACK are written into the file ~/STACK.T. The buffer STACK is linked to the file ~/STACK.T (see 1.2 [On Emacs], page 3 for more information). Note that the parser of the OBSCURE system expects the filenames given as argument to end in .T.

MINIBUFFER:

```
Wrote /users/arbor/STACK.T
```

The user can now complete the specification text by editing the buffer STACK with the help of the *Emacs* editing commands. The buffer now looks as follows:

```
---------------------------- STACK ----------------------------

IMPORTS
 SORTS el

CREATE
 SORTS stack

 OPNS
  empty:  -> stack
   _ push _ :  el, stack -> stack
  top:  stack -> el
  pop:  stack -> stack

SEMANTICS
 CONSTRS

 VARS

 PROGRAMS
```

```
ENDCREATE
                                STACK
```

The list of operations following the keyword *OPNS* is copied behind the keyword *CONSTRS* to save work.

INPUT: `C-o c` # call of the function `M-x o-copy-opns`

```
                                STACK

IMPORTS
 SORTS el

CREATE
 SORTS stack

 OPNS
  empty:  -> stack
  _ push _ :  el, stack -> stack
  top:  stack ->el
  pop:  stack -> stack

SEMANTICS
 CONSTRS
  empty:-> stack
  _ push _ :  el, stack -> stack
  top:  stack -> el
  pop:  stack -> stack

 VARS
 PROGRAMS

ENDCREATE
                                STACK
```

Operations that are not constructors are now deleted from the list with the help of *Emacs* editing commands.

The text is modified by further editing, until it is of the following form:

```
                              STACK
IMPORTS
 SORTS el

CREATE
 SORTS stack

 OPNS
  empty:  -> stack
  _ push _ :  el, stack -> stack
  top:  stack ->el
  pop:  stack -> stack

SEMANTICS
 CONSTRS
  empty:-> stack
  _ push _ :  el, stack -> stack

 VARS e:  el,
    s, s':  stack

 PROGRAMS
  top(s) <- CASE s OF

ENDCREATE
                              STACK
```

The constructors of the sort `stack` are needed to define the recursive program for the operation `top`. The OBSCURE system offers a command which displays the constructors of a sort `s` of the specification in the current buffer.

INPUT: `C-o h` # call of the function `M-x o-show-constrs`

MINIBUFFER:

`new sort:`

INPUT: `stack`<RET>

The screen is divided into two windows. The upper one still displays the contents of the buffer `STACK`, the lower one displays the contents of a new buffer called `*Constructors*`:

```
*Constructors*

Constructors for the new sort stack:

   empty:  -> stack
   _ push _ :  el, stack -> stack

*Constructors*
```

MINIBUFFER:

```
Type C-x 1 to remove *Constructors* window
```

With the information of the buffer ***Constuctors*** the specification of a stack of elements can now be completed. The buffer **STACK** contains the following text:

```
STACK

IMPORTS
 SORTS el

CREATE
 SORTS stack
 OPNS
  empty:  -> stack
  _ push _ :  el, stack -> stack
  top:  stack ->el
  pop:  stack -> stack

SEMANTICS
 CONSTRS
  empty:-> stack
  _ push _ :  el, stack -> stack

 VARS e:  el,
s, s':  stack

 PROGRAMS
  top(s) <- CASE s OF
             empty:  null;
             e push s':  e
            ESAC;
    pop(s) <- CASE s OF
             empty:  empty;
             e push s':  s'
             ESAC
ENDCREATE

STACK
```

INPUT: `C-x 1` # Removal of the second text window

The window displaying the contents of the buffer `*Constructors*`, disappears from the screen.

2.3 Using the parser, correcting errors

To illustrate the correction of errors an error has been included into the specification of the last section:

```
top(empty)=null
```

The operation `null: -> el` is unknown to the parser, because it has been declared neither as imported nor as created.

INPUT: `C-o p` # call of the function `M-x o-parser`

This command starts the OBSCURE parser.

MINIBUFFER:

```
Buffer name: STACK
```

INPUT: <RET>

MINIBUFFER:

```
Options [cfv]:
```

INPUT: <RET>

MINIBUFFER:

```
Save file /users/arbor/STACK.T? (y or n)
```

INPUT: y

MINIBUFFER:

```
Wrote /users/arbor/STACK.T
```

MINIBUFFER:

```
Save file /users/arbor/STACK.T? (y or n)
```

INPUT: y

MINIBUFFER:

```
Wrote /users/arbor/STACK.T
```

The screen is divided into two windows. The upper one still displays the buffer **STACK**, the lower one displays the new buffer ***compilation*** which, after parsing, contains the following text:

```
---------------------- *compilation* ----------------------
cd /users/arbor/
/users/obscure/d-run/compile-command /users/arbor/STACK.T
STACK.T, line 23:  token:  "null";
here the name "null" is defined neither as prefixname nor as
infixname nor as mixfixname nor as variablename
STACK.T, line 23:
token:  "null"; syntax error
specification is not accepted:
1 errors

Compilation exited abnormally with code 1 at Wed Aug 16 12:55:57
---------------------- *compilation* ----------------------
```

The error in the specification has been found and described by the parser.

INPUT: `C-x‘` # call of the function: `M-x next-error`

The cursor is positioned on column 1 in line 23 (viz. the line containing the error) in the buffer STACK.

The specification is now modified in order to correct this error: the word `null` is replaced by the predefined operation `ERROR(el)`. In the case construct of the operator pop the case of the empty stack is redefined to `ERROR(stack)`. The buffer now looks as follows:

```
------------------------------------ STACK ------------------------------------
IMPORTS
 SORTS el

CREATE
 SORTS stack
 OPNS
  empty:  -> stack
  _ push _ :  el, stack -> stack
  top:  stack ->el
  pop:  stack -> stack

SEMANTICS
 CONSTRS
  empty:-> stack
  _ push _ :  el, stack -> stack

 VARS e:  el,
s, s':  stack

 PROGRAMS
  top(s) <- CASE s OF
             empty:  ERROR(el);
             e push s':  e
            ESAC;
    pop(s) <- CASE s OF
             empty:  ERROR(stack);
             e push s':  s'
             ESAC
ENDCREATE
------------------------------------ STACK ------------------------------------
```

INPUT: `C-o p` # call of the function `M-x o-parser`

The buffer `*compilation*` now displays the following:

```
--------------------------------- *compilation* ---------------------------------
cd /users/arbor/
/users/obscure/d-run/compile-command  /users/arbor/STACK

Compilation finished at Wed Aug 16 13:11:23
--------------------------------- *compilation* ---------------------------------
```

The specification `STACK` has been parsed correctly; it has been automatically entered into the module database of the user `arbor`. This command is the only one allowing to enter specifications into the user's module database; it guarantees that the module database contains syntactically correct specifications only. The lower window is now removed by the following input:

INPUT: `C-x 1` # removing the lower textwindow

2.4 Using the module database

The following sections explain the use of the user's personal module database and the global database. This is accomplished by resuming the example from the last sections. A database query concerning the imported and exported sorts and operations and the exported constructors of the module **STACK** is started as follows:

INPUT: `C-o s` # call of the function **o-mdb-select**

A buffer called ***Data-base-input*** is displayed on the screen.

```
------------------------------ *Data-base-input* ------------------------------
**************************** Select from MDB ********************************
  C-i start the search in the data base
  C-c start selecting field names    ESC escape from selecting field names
  RET select a field name            C-t toggle all fields
  SPC move to the next field name    DEL move to the previous field name
*****************************************************************************
SELECT FROM
 Data base:  arbor
 List of field names:  mname
      iparams [ ] eparams [ ] isorts  [ ] esorts   [ ]
      iopns   [ ] eopns   [ ] iconstr [ ] econstr  [ ]
      iaxioms [ ] eaxioms [ ] uses    [ ] isusedby [ ]   compiled [ ]
WHERE Condition:

------------------------------ *Data-base-input* ------------------------------
```

This buffer can now be edited with the commands offered in the upper part of the window. The condition for the search in the database (`mname = "STACK"`) has to be written via *Emacs* editing commands onto the last line behind the words "`WHERE Condition:`". `mname` is a keyword and stands for the name of the module; the name of the module itself (= `STACK`) must be written in double quotes (`"STACK"`). After editing the buffer should look as follows:

```
----------------------------- *Data-base-input* ------------------------------

**************************** Select from MDB ********************************
  C-i start the search in the data base
  C-c start selecting field names      ESC escape from selecting field names
  RET select a field name              C-t toggle all fields
  SPC move to the next field name      DEL move to the previous field name
*****************************************************************************
SELECT FROM
 Data base:  arbor
 List of field names:  mname
      iparams [ ] eparams [ ] isorts  [X] esorts   [X]
      iopns   [X] eopns   [X] iconstr [X] econstr  [X]
      iaxioms [ ] eaxioms [ ] uses    [ ] isusedby [ ]   compiled [ ]
WHERE Condition:  mname = "STACK"

----------------------------- *Data-base-input* ------------------------------
```

The search is then initiated by typing the following:

INPUT: `C-i` # call of the function `M-x o-make-mdb-input`

The contents of the buffer `STACK` are displayed in the upper half of the screen. The new buffer `*MDB-OUTPUT*` containing the answer to the database query is displayed in the lower half.

```
--------------------------- *MDB-OUTPUT* ---------------------------
Start
arbor.mname:
        STACK
arbor.isorts:
        el
arbor.esorts:
        el
        stack
arbor.iopns:
        (nil)
arbor.eopns:
        pop:stack->stack
        top:stack->el
arbor.iconstr:          (nil)

arbor.econstr:
        _ push _:el,stack->stack
        empty:->stack

## The End
--------------------------- *MDB-OUTPUT* ---------------------------
```

A stack of natural numbers is specified in the following text. In order to find out whether such a specification already exists the global module database is checked. (The careful reader of the OBSCURE manual may have realized that the integers are a predefined sort ("basic sort") of the OBSCURE system (see C [The syntax for the specification language], page 96 for more information). In contrast the natural numbers are not predefined.) The second textwindow is removed first:

INPUT: `C-x 1` # removal of the second textwindow

The command `C-o s` is used to find out whether there is a specification module of natural numbers in the database. To be more specific, the user investigates whether there is a specification name containing 'nat' as a substring. This is expressed by writing `"LIKE"` instead of "=".

INPUT: `C-o s` # call of the function `M-x o-mdb-select`

As above the buffer *Data-base-input* is edited:

```
                        *Data-base-input*

*************************** Select from MDB ******************************
  C-i start the search in the data base
  C-c start selecting field names    ESC escape from selecting field names
  RET select a field name            C-t toggle all fields
  SPC move to the next field name    DEL move to the previous field name
**************************************************************************

SELECT FROM
 Data base:  mdbpool
 List of field names:  mname
     iparams [ ] eparams [ ] isorts  [X] esorts   [X]
     iopns   [X] eopns   [X] iconstr [X] econstr  [X]
     iaxioms [ ] eaxioms [ ] uses    [ ] isusedby [ ]   compiled [ ]
WHERE Condition:  mname LIKE "nat"
                        *Data-base-input*
```

The search is then started:

INPUT: C-i # call of the function M-x o-make-mdb-input

The result of the search is displayed in the buffer *MDB-OUTPUT*

```
                        *MDB-OUTPUT*

## Start
## The End
                        *MDB-OUTPUT*
```

The answer is negative. Hence it is improbable that the natural numbers have already been specified. Therefore a specification NAT is created in the same way as the specification STACK. For the creation of STACK, repeat all neccessary steps until the buffer NAT is of the following form:

```
                        NAT

CREATE
 SORTS
  nat

 OPNS
  add :  nat nat -> nat
  succ :  nat -> nat
  null :  -> nat

SEMANTICS
 CONSTRS
  succ :  nat -> nat
  null :  -> nat
```

```
 VARS i1 i2 i1' :  nat

 PROGRAMS
  add(i1, i2) <- CASE i1 OF
                  null:  i2;
                  succ(i1'):  succ(add(i1', i2));
                 ESAC;
ENDCREATE
```
NAT

The specification is then parsed.

INPUT: `C-o p` # call of the function `M-x o-parser`

The window of the buffer `*compilation*` now displays:

compilation
```
cd /users/arbor/
/users/obscure/d-run/compile-command  /users/arbor/NAT

Compilation finished at Wed Aug 16 13:11:23
```
compilation

In order to control the entry into the database, the text of the specification `NAT` is read from the database.

INPUT: `C-o w` # call of the function `M-x o-print-spec`

MINIBUFFER:

`Module name:`

INPUT: `NAT`<RET>

MINIBUFFER:

`Data base:  mdbpool`

The word `mdbpool` has to be deleted and replaced by `arbor`, because `NAT` does not exist in the global module database, but in the personal module database (of the user, whose name is `arbor`).

INPUT: <DEL><DEL><DEL><DEL><DEL><DEL><DEL>

INPUT: arbor<RET>

MINIBUFFER:

```
Options [scvrb]:
```

INPUT: <RET>

The screen is divided and in the lower half the buffer ***Specification*** is displayed.

```
---------------------- *Specification* ----------------------

IMPORTS
 ## no sorts
 ## no operations
CREATE
 SORTS
  nat
 OPNS
  add :  nat nat -> nat
  succ :  nat -> nat
  null :  -> nat
SEMANTICS
 CONSTRS
  succ :  nat -> nat
  null :  -> nat
 VARS
  i1 i2 i1' :  nat
 PROGRAMS
  add(i1, i2) <-
   CASE i1
   OF
    null:
     i2;
    succ(i1'):
     succ(add(i1', i2));
   ESAC;


ENDCREATE
## The End
---------------------- *Specification* ----------------------
```

MINIBUFFER:

```
Type C-x 1 to remove window
```

INPUT: C-x 1

The window displaying the contents of the buffer **NAT** now fills the whole screen.

2.5 Building composed specifications

We are now ready to build the specification of a stack of natural numbers `NAT-STACK` from these two specifications.

INPUT: `C-x b` # changing to a new buffer

MINIBUFFER:

```
Switch to buffer: (default *Specification*)
```

INPUT: `NAT-STACK`<RET>

This buffer being in the **fundamental-mode** it has to be switched into the **obscure-mode**.

INPUT: `M-o` # switch to **obscure-mode**

The empty buffer `NAT-STACK` is displayed on the screen. After editing it should contain the following specification text:

NAT-STACK

```
(INCLUDE STACK
 I_RENAME SORTS el AS SORTS nat)
X_COMPOSE
(INCLUDE NAT)
```

NAT-STACK

Before this file is parsed, it is saved with the *Emacs* command `C-x C-s`.

INPUT: `C-x C-s` # call of the function `M-x save-buffer`

MINIBUFFER:

```
File to save in:~/
```

INPUT: `NAT-STACK.T`<RET>

MINIBUFFER:

```
Wrote /users/arbor/NAT-STACK.T
```

INPUT: `C-o p` # call of the function `M-x o-parser`

MINIBUFFER:

```
File name:~/NAT-STACK
```

INPUT: <RET>

MINIBUFFER:

```
Options [cfv]:
```

INPUT: <RET>

MINIBUFFER:

```
Save file /users/arbor/NAT-STACK.T? (y or n)
```

INPUT: y

MINIBUFFER:

```
Wrote /users/arbor/NAT-STACK.T
```

The buffer `*compilation*` then contains the following text:

```
------------------------ *compilation* ------------------------
cd /users/arbor/
/users/obscure/d-run/compile-command  /users/arbor/NAT-STACK.T

Compilation finished at Wed Aug 16 15:26:22
------------------------ *compilation* ------------------------
```

The specification of a stack of natural numbers is now completed.

2.6 Using the module database (continued)

The personal module database now contains the specification of a stack of natural numbers. Some of the possibilities to get information about composed specifications are illustrated in the following.

The signature of the specification is displayed by the command:

INPUT: `C-o t` # call of the function `M-x o-curr-signature`

The window displaying the buffer `*compilation*` is replaced by a window showing the new buffer `*Signature*`:

```
*Signature*

File name:  "/users/arbor/NAT-STACK.T"
list of sorts and operations:

imported and exported (inherited).
  ## no sorts
  ## no operations

not imported and exported (created).
 SORTS
  stack nat
 OPNS
  _ push _ :  nat stack -> stack
  pop :  stack -> stack
  empty :  -> stack
  add :  nat nat -> nat
  succ :  nat -> nat
  null :  -> nat
  top :  stack -> nat

imported and not exported (hidden).
  ## no sorts
  ## no operations

list of constructors:

imported

exported
  null :  -> nat
  succ :  nat -> nat
  empty :  -> stack
  _ push _ :  nat stack -> stack
## The End

*Signature*
```

As explained earlier, the command `C-o w` is used to read the text of a specification from the database. It can also be used to instantiate the calls of the specifications `STACK` and `NAT` by their actual texts. The option `"-r"` is used for this purpose.

INPUT: `C-o w` # call of the function `M-x o-print-spec`

MINIBUFFER:

```
Module name:  NAT
```

INPUT: <DEL><DEL><DEL>`NAT-STACK`<RET>

MINIBUFFER:

```
Data base:  arbor
```

INPUT: <RET>

MINIBUFFER:

```
Options [scvrb]:
```

INPUT: -r<RET>

The contents of the buffer `*Specification*` are displayed in the lower half of the screen.

```
*Specification*
(
(
IMPORTS
 SORTS
  el
  ## no operations
CREATE
 SORTS
  stack
 OPNS
  _ push _ :  el stack -> stack
  pop :  stack -> stack
  empty :  -> stack
  top :  stack -> el
SEMANTICS
 CONSTRS
  _ push _ :  el stack -> stack
  empty :  -> stack
 VARS
  s' s :  stack
  e :  el
 PROGRAMS
  pop(s) <-
   CASE s
   OF
    empty:
     ERROR(stack);
```

```
    e push s':
     s';
   ESAC;

  top(s) <-
   CASE s
   OF
    empty:
     ERROR(el);
    e push s':
     e;
   ESAC;

ENDCREATE
)
I_RENAME
 SORTS
  el
  ## no operations
AS
 SORTS
  nat
  ## no operations
)
X_COMPOSE
(
(
IMPORTS
 ## no sorts
 ## no operations
CREATE
 SORTS
  nat
 OPNS
  add :  nat nat -> nat
  succ :  nat -> nat
  null :  -> nat
SEMANTICS
 CONSTRS
  succ :  nat -> nat
  null :  -> nat
 VARS
  i1 i2 i1' :  nat
 PROGRAMS
  add(i1, i2) <-
   CASE i1
   OF
     null:
```

```
      i2;
    succ(i1'):
      succ(add(i1', i2));
  ESAC;


ENDCREATE
)
)
## The End
```

Specification

In order to work with this specification it is saved in a file. The buffer called `*Specification*` is made the active buffer first; note that the name `*Specification*` can be typed as `*Sp`<TAB> (because *Emacs* tries to complete the name when <TAB> is entered).

INPUT: `C-x b`

MINIBUFFER:

```
Switch to buffer:  (default *Signature*)
```

INPUT: `*Sp`<TAB><RET>

INPUT: `C-x C-s` # call of the function `M-x save-buffer`

MINIBUFFER:

```
File to save in ~/
```

INPUT: `NAT-STACK2.T`<RET>

MINIBUFFER:

```
Wrote /users/arbor/NAT-STACK2.T
```

The text of the specification `NAT-STACK`, in which the instantiations of the modules `NAT` and `STACK` have been replaced by their text, has been saved in the file `NAT-STACK2.T`, by the sequence of commands used above. The module `NAT-STACK` (from the module database) and the text in the file `NAT-STACK2.T` have the same semantic meaning; they only differ in the text of their specification. By using the parser it is shown that `NAT-STACK2` is a syntactically correct specification, too. Before the parser can be called the buffer has to be switched to the `obscure-mode` (this time the rather long word `obscure-mode` is abbreviated by the name completion as explained above).

INPUT: `M-x ob`<TAB><RET> # call of the function `M-x obscure-mode`

INPUT: `C-o p` # call of the function `M-x o-parser`

MINIBUFFER:

```
File name: *Specification*
```

INPUT: <RET>

MINIBUFFER:

```
Options [cfv]:
```

INPUT: <RET>

MINIBUFFER:

```
Save file /users/arbor/NAT-STACK2.T? (y or n)
```

INPUT: y

MINIBUFFER:

```
Wrote /users/arbor/NAT-STACK2.T
```

```
------------------------ *compilation* ------------------------

cd /users/arbor/
/users/obscure/d-run/compile-command  /users/arbor/NAT-STACK2.T

Compilation finished at Wed Aug 16 16:32:57
------------------------ *compilation* ------------------------
```

In order to compare the two specification modules NAT-STACK and NAT-STACK2 a database query is started.

INPUT: C-x 1 # removal of the lower textwindow

INPUT: C-o s # call of the function M-x o-mdb-select

The well-known buffer of the database query appears. It is edited until it looks as follows:

```
------------------------------ *Data-base-input* ------------------------------

*************************** Select from MDB ******************************
  C-i  start the search in the data base
  C-c  start selecting field names    ESC  escape from selecting field names
  RET  select a field name            C-t  toggle all fields
  SPC  move to the next field name    DEL  move to the previous field name
**************************************************************************

SELECT FROM
 Data base:  arbor
 List of field names:  mname
      iparams [ ]  eparams  [ ]  isorts  [X]  esorts    [X]
      iopns   [X]  eopns    [X]  iconstr [ ]  econstr   [X]
      iaxioms [ ]  eaxioms  [ ]  uses    [X]  isusedby [ ]   compiled [ ]
WHERE Condition:  mname = "NAT-STACK2"

------------------------------ *Data-base-input* ------------------------------
```

INPUT: C-i # call of the function M-x o-make-mdb-input

```
------------------------------- *MDB-OUTPUT* -------------------------------

## Start
arbor.mname:
        NAT-STACK2
arbor.isorts:
        (nil)

arbor.esorts:
        stack
        nat
arbor.iopns:
        (nil)

arbor.eopns:
        pop:stack->stack
        add:nat,nat->nat
        top:stack->nat
arbor.econstr:
        _ push _:nat,stack->stack
        empty:->stack
        succ:nat->nat
```

```
        null:->nat
arbor.uses:
        (nil)

## The End
                          *MDB-OUTPUT*
```

It is now clear that NAT-STACK and NAT-STACK2 specify the same sorts and operations. Therefore, the specification NAT-STACK2 is deleted from the module database arbor with the following steps.

INPUT: C-o d # call of the function M-x o-mdb-delete

MINIBUFFER:

```
specification name:
```

INPUT: NAT-STACK2<RET>

MINIBUFFER:

```
data base name:  arbor
```

INPUT: <RET>

MINIBUFFER:

```
Do you want a protocol of the deletions ?  (y or n)
```

INPUT: y

A protocol of the executed work is shown in the buffer *MDB-OUTPUT* in the lower textwindow.

```
                          *MDB-OUTPUT*
Protocol of deletions:

Relation NAT-STACK2 deleted in database "arbor".
File NAT-STACK2.O deleted.
## The End

                          *MDB-OUTPUT*
```

2.7 Using the interpreter

In the following section the usage of the OBSCURE interpreter is explained with the help of the specification of natural numbers created so far. The interpreter is started by typing:

INPUT: `C-o i` # call of the function `M-x o-interprete`

The prompt of the interpreter `;-)` appears in the buffer `*Interpreter*` in the lower textwindow:

```
---------------------- *Interpreter* ----------------------
;-)

---------------------- *Interpreter* ----------------------
```

A specification is loaded into the interpreter by typing the following command:

INPUT: `currspec NAT-STACK`

The command `currspec` makes the specification `NAT-STACK` known to the interpreter. The buffer `*Interpreter*` now looks as follows:

```
---------------------- *Interpreter* ----------------------
;-) currspec NAT-STACK
;-)
---------------------- *Interpreter* ----------------------
```

The interpreter is ready for new inputs. If the command `currspec` is called without a name as parameter, then the name of the current specification is displayed.

Names and bindings can be made known to the interpreter by the command `let NAME=TERM`. Names must start with the symbol `$`. Some names will be made known to the interpreter for further experiments.

INPUT: `let $one=succ(null)`

INPUT: `let $two=succ(succ(succ(null)))`

INPUT: `let $three=succ(succ(succ(null)))`

It is possible to show the binding of a name or of all names by the commands `show NAME` or `show`, respectively.

INPUT: `show`

The bindings of all names are shown in the buffer `*Interpreter*`.

```
------------------------- *Interpreter* -------------------------
;-) show
name:  "$three    "  value:  "succ(succ(succ(null)))"
name:  "$one      "  value:  "succ(null)"
name:  "$two      "  value:  "succ(succ(succ(null)))"
;-)
------------------------- *Interpreter* -------------------------
```

During the input of `$two` a mistake occurred. This mistake is corrected with the help of the command `delete NAME` which deletes the binding of `NAME` (`delete` without the parameter `NAME` deletes every binding).

INPUT: `delete $two`

INPUT: `let $two=succ(succ(null))`

INPUT: `show $two`

The buffer `*Interpreter*` now looks as follows:

```
------------------------- *Interpreter* -------------------------
;-) delete $two
;-) let $two =succ(succ(null))
;-) show $two
value:  "succ(succ(null))"
;-)
------------------------- *Interpreter* -------------------------
```

It is possible to save bindings in a file for further usage. This process is demonstrated with the current bindings.

INPUT: `write nat` # saving the bindings in the file nat.B

In order to check the success of the saving and in order to show the loading of bindings, all bindings are deleted and the bindings just saved are loaded into the system and listed.

INPUT: `delete`<RET> # deleting all bindings

INPUT: `y`<RET>

INPUT: `read nat`<RET> # loading the bindings from the file nat.B

INPUT: show<RET> # displaying the bindings

After this sequence of commands the buffer *Interpreter* looks as follows:

```
*Interpreter*

;-) write nat
;-) delete
really delete all bindings (y/n) y
;-) read nat
;-) show
name:  "$three     "  value:  "succ(succ(succ(null)))"
name:  "$one     "  value:  "succ(null)"
name:  "$two     "  value:  "succ(succ(null))"
;-)

*Interpreter*
```

We now shortly illustrate the evaluation of terms.

First, the result of add($three,$two) is evaluated and bound to the name $five. Then, a stack of the elements $two, $three and $five is created and bound to the name $stack. Finally, the first element of this stack is listed before and after an application of pop.

INPUT: eval add($three,$two)<RET>

INPUT: let $five=add($three,$two)<RET>

INPUT: let $stack=$five push ($three push ($two push empty))

INPUT: <RET>

INPUT: eval $stack<RET>

INPUT: eval top($stack)<RET>

INPUT: eval top(pop($stack))<RET>

During the input the following appears in the buffer *Interpreter*:

```
*Interpreter*

;-) eval add($three, $two)
succ(succ(succ(succ(succ(null)))))
;-) let $five=add($three, $two)
;-) let $stack =$fuenf push ($three push ($two push empty))
;-) eval $stack
succ(succ(succ(succ(succ(
null))))) push succ(succ(succ(null)))push succ(succ(null))
push empty
;-) eval top($stack)
succ(succ(succ(succ(succ(null)))))
```

```
;-) eval top(pop($stack))
succ(succ(succ(
null)))
;-)
```

Interpreter

The session with the interpreter is stopped by typing the quit command.

INPUT: quit<RET>

The interpreter of the OBSCURE system offers further possibilities, such as a trace–mode used for debugging incorrect specifications. For more information see 4.4 [The Interpreter], page 66.

2.8 Translating into the programming language C

The OBSCURE system offers the possibility to translate specifications into the programming language C. For the moment this is only possible for atomic specifications. The process of a translation is illustrated in the following. More precisely, the specification NAT used in the examples of the previous sections is translated into C. To this end the buffer containing the specification of the natural numbers must be changed into the active buffer.

INPUT: C-x b

MINIBUFFER:

```
Switch to buffer:  (default NAT-STACK)
```

INPUT: NAT<RET>

Then, the Source–to–Source–Translation (So–to–So–Tra for short) is started as follows:

INPUT: C-o u # call of the function M-x o-So-To-So-Tra

MINIBUFFER:

```
Specification name:  NAT
```

INPUT: <RET>

MINIBUFFER:

```
Option [c,m]:  c                                # c für C, m für ML
```

INPUT: <RET>

The screen splits into two windows. The upper one displays a buffer called `NAT.h` that looks as follows:

NAT.h

```
enum SOT_nat {SOT_succ, SOT_null};

struct nat {
  enum SOT_nat SOT_Typ;
  union {
    struct succ {
      struct nat *SOT_1;
    } succ;
  } SOT_union;
};

struct nat *add();
struct nat *succ();
struct nat *null();
```

NAT.h

The lower window displays a buffer called `NAT.c` that looks as follows:

NAT.c

```
#include<stdio.h>
#include "NAT.h"

#define true 1
#define false 0

struct nat *nat_copy(arg)
  struct nat *arg;
{
return(
arg->SOT_Typ==SOT_succ?succ(nat_copy(arg->SOT_union.succ.SOT_1)):
arg->SOT_Typ==SOT_null?null():
NULL);
}

struct nat *succ(v1)
  struct nat *v1;
{
  struct nat *ret;

  ret=(struct nat *)malloc(sizeof(struct nat));
  ret->SOT_union.succ.SOT_1=v1;
  ret->SOT_Typ=SOT_succ;
  return(ret);
```

```
}

struct nat *null()
{
  struct nat *ret;

  ret=(struct nat *)malloc(sizeof(struct nat));
  ret->SOT_Typ=SOT_null;
  return(ret);
}

struct nat *add(i1, i2)
  struct nat *i1;
  struct nat *i2;
{
return(
(i1->SOT_Typ==SOT_succ?succ(add(i1->SOT_union.succ.SOT_1, i2)):
i1->SOT_Typ==SOT_null?i2:
NULL)
);
}
```

NAT.c

These two buffers can be saved by typing `C-x s`. The files thus created can then be compiled with a C compiler (after adding a '`main`' function).

2.9 Ending the session

The session with the OBSCURE system is stopped by the following input:

INPUT: `C-x C-c`

************************ Return to the shell *****************************

Chapter 3

The specification language OBSCURE

This chapter merely constitutes a rough description of the constructs of the specification language OBSCURE. For a precise description of the specification language the reader is referred to [LL 88] or [LL 90], and for a precise description of the algorithmic specification method to [Lo 87].

Essentially the specification language OBSCURE consists of atomic specifications drawn up according to the algorithmic specification method together with constructs which allow to put specifications together.

An atomic specification consists of a list of imported sorts and operations and/or a list of created sorts and operations. Some of the created operations are declared as *constructors*. The semantics of a created sort is defined to be the term language generated by the constructors. The semantics of the operations are described in the form of recursive programs.

Four constructs allow to compose specifications: *PLUS*, *COMPOSE*, *X_COMPOSE* and *F_COMPOSE*. The latter two may be viewed as macros.

Five further constructs allow to rename imported and exported sorts and operations, to add axioms restricting the class of imported and exported sorts and operations, and to forget sorts and operations (*information hiding*). Finally, two more constructs allow to build subalgebras and quotient algebras.

Semantically an OBSCURE-specification describes an algebra module, i.e. a function mapping algebras (of the imported signature) into algebras (of the exported signature). The exact definition of the semantics and the proof that an OBSCURE-specification indeed defines an algebra module can be found in [LL 88] and [LL 90].

The full syntax of OBSCURE can be found in Appendix C (See C [The context free syntax for the specification language], page 96, for more info).

A formal description of the context conditions of OBSCURE can be found in [Zey 89].

3.1 Atomic specifications

3.1.1 General

Syntactically, an atomic specification consists of three parts. The first part is a list of "imported" sorts and operations and is introduced by the keyword *IMPORTS*. The second part is a list of "new" sorts and operations and is introduced by the keyword *CREATE*. The third part associates a semantic with the new sorts and operations and is introduced by the keyword *SEMANTICS*. The semantics are defined by a list of "constructors" (*CONSTRS*) and a list of recursive programs (*PROGRAMS*). Operations are defined as prefix, infix or mixfix operations. A context free syntax for atomic specifications can be found in section 3.1.2.

Typical context-conditions are:

- prefix operations with the same name must differ by the number of their arguments or the sort of at least one of their arguments;
- the name of an infix operation may not appear as one of the component names of a mixfix operation;
- the variables occurring in a recursive program must be among those listed after *VARS*;
- there is exactly one recursive program for each new operation that is not a constructor.

A complete list of these context conditions may be found in [Zey 89].

The semantics of atomic specifications is precisely described in [Lo 87]. Note the two following additional facilities used in OBSCURE. *LAZY* allows the introduction of infinite carriers (and lazy interpretations of corresponding operations). By importing constructors (*WITH IMPORTED CONSTRS*) it is possible to use case distinction on the basis of constructors specified elsewhere.

3.1.2 A context free syntax

The syntax is written in Extended Backus-Naur Form: [...] means that the part within [and] is optional and {...} means that the part within { and } may be omitted or repeated.

```
<atomic specification> ::= IMPORTS <list of sorts and operations>
                         | IMPORTS <list of sorts and operations>
                           CREATE <list of sorts and operations>
                           SEMANTICS <algorithmic semantics>
                           ENDCREATE
                         | CREATE
                           <list of sorts and operations>
                           SEMANTICS <algorithmic semantics>
                           ENDCREATE

<list of sorts and operations> ::= SORTS <list of sorts>
                                 | SORTS <list of sorts>
                                   OPNS <list of operations>
                                 | OPNS <list of operations>

<list of sorts>                ::= { <sort>[,] }

<sort>                         ::= <name>

<list of operations>           ::= { <operation>[,] }

<operation>                    ::= <name>:-> <sort>
                                 | <name>: <sort> {[,] <sort>} -> <sort>
                                 | <name> _: <sort> -> <sort>
                                 | _ <name> _: <sort>[,] <sort>
                                     -> <sort>
                                 | <name> _ <name> { _ <name> }:
                                       <sort> {[,] <sort> } -> <sort>
                                 | _ <name> { _ <name> }:
                                       <sort> {[,] <sort> } -> <sort>

<name>                         ::= <letter> { <symbol> }
                                 | <special symbol>

<letter>                       ::= a | b | c | ... | z | A | B | ... | Z

<symbol>                       ::= <letter> | <digit> | _
                                 | <special symbol>

<digit>                        ::= 0 | 1 | 2 | ... | 9
```

```
<special symbol>             ::=  < | > | = | # | ^ | [ | ] |   |   | +
                                 | * | / | - |   | @ | $ | %

<algorithmic semantics> ::=  CONSTRS <list of constructors>
                             [ [ WITH IMPORTED CONSTRS
                                 <list of constructors> ]
                              VARS <list of variables>
                              PROGRAMS <list of programs> ]
                           | [ WITH IMPORTED CONSTRS
                             <list of constructors> ]
                             VARS <list of variables>
                             PROGRAMS <list of programs>

<list of variables>     ::=  {{ <name>[,] }: <sort>[,] }

<list of constructors>  ::=  { <constructor>[,] }

<constructor>           ::=  <name>:-> <sort>
                           | <name>: <args> {[,] <args> } -> <sort>
                           | <name> _: <args> -> <sort>
                           | _ <name> _: <args>[,] <args> -> <sort>
                           | <name> _ <name> {[,] <name> }:
                                <args> {[,] <args> } -> <sort>
                           | _ <name> { _ <name> }:
                                <args> {[,] <args> } -> <sort>

<args>                  ::=  [ LAZY ] <sort>

<list of programs>      ::=  <head> <- <term>; { <head> <- <term>; }

<head>                  ::=  <prefix name>
                           | <prefix name> ({ <variable>, })
                           | <infix name> <variable>
                           | <variable> <infix name> <variable>
                           | <mixfix name> <variable> <mixfix name>
                             { <variable> < mixfix name> }
                           | <variable> <mixfix name>
                             { <variable> <mixfix name> }

<prefix name>           ::=  <name>

<infix name>            ::=  <name>

<mixfix name>           ::=  <name>

<variable>              ::=  <name>
```

```
<term>              ::=  <infixterm>
                       | <mixfix name> <infixterm> <mixfix name>
                         { <infixterm> <mixfix name> }

<infixterm>         ::=  <baseterm>
                       | <infixterm> <infix name> <baseterm>

<baseterm>          ::=  (<term>)
                       | ERROR (<sort>)
                       | <variable>
                       | IF <term> THEN <term>
                         ELSE <term> FI
                       | CASE <term> OF
                         <head>: <term>; { <head>: <term>; };
                         [ ELSE <term> ] ESAC
                       | <prefix name>
                       | <prefix name> (<term< { , <term> })
                       | <infix name> <baseterm>
```

3.1.3 A simple example

The following text is a simple example specifying a list of elements. *ERROR* stands for the undefined value (of [Lo 87]).

```
IMPORTS
 SORTS el
 OPNS _ = _: el el -> bool

CREATE
 SORTS list
 OPNS nil              : -> list
      _ _             : list el -> list
      _ is_empty       : list -> bool
      last_of _        : list -> el
      body_of _        : list -> list
      append _ to _ end : list list -> list

SEMANTICS
 CONSTRS
  nil    : -> list
  _ _   : list el -> list

 VARS
  e e' e'' : el
  l l' l'' : list
```

```
PROGRAMS
l is_empty          <- l = nil;

last_of l           <- CASE l OF
                         nil   : ERROR(list);
                         l' e : e
                       ESAC;

body_of l           <- CASE l OF
                         nil    : ERROR(list);
                         l' e : l'
                       ESAC;

append l to l' end <- CASE l OF
                         nil     : l';
                         l'' e : append l'' to (l' e) end
                       ESAC;
ENDCREATE
```

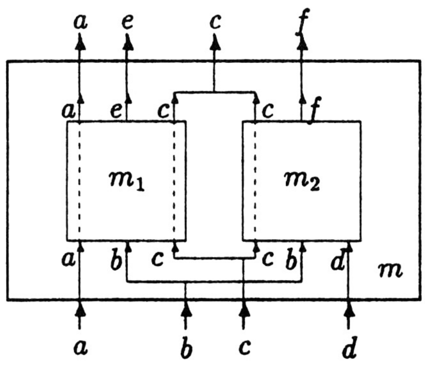
3.2 The constructs

3.2.1 The compose constructs

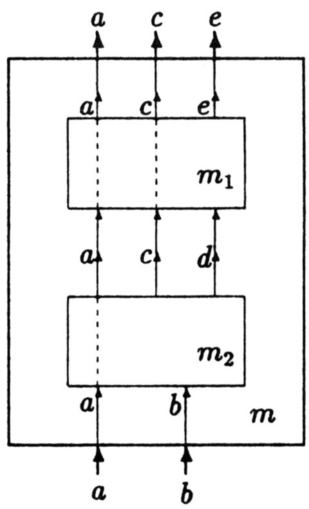
The constructs *PLUS* and *COMPOSE* are the constructs $+$ and $\circ$ of [LL 88] and [LL 90]. For a precise definition of their semantics the reader is refered to these papers. A graphical illustration is on Figure 1 below.

The constructs *X_COMPOSE* and *F_COMPOSE* allow to avoid the stringent context conditions of the *COMPOSE* construct. They are illustrated on Figure 2 on the next page.

The four constructs may be illustrated by the following figures.

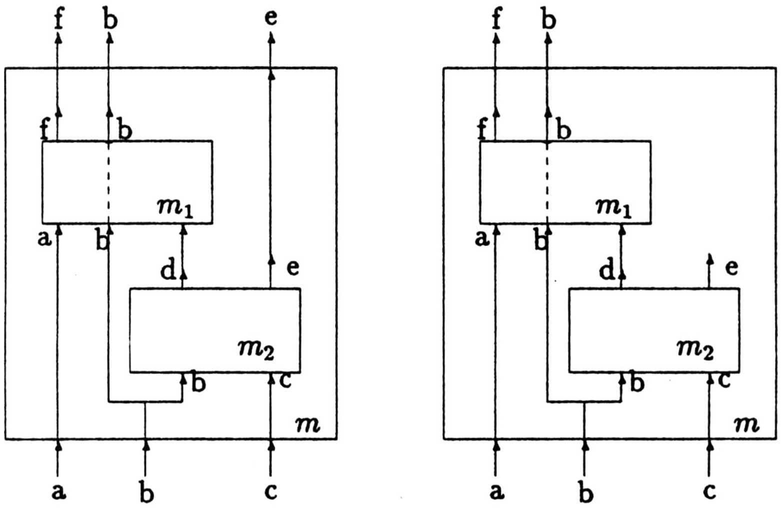


(a) $m = (m_1 + m_2)$



(b) $m = (m_1 \circ m_2)$

FIGURE 1 Graphical illustration of the compose constructs of the specification language. In this illustration a specification is represented by a box. The arrows entering a box represent its imported sorts and operations, those leaving a box represent its exported sorts and operations. A dotted line represents an inherited sort or operation. Each of the symbols $a, b, \ldots, e, f$ stands for a sort or an operation.



(c) $m = (m_1$ **X_COMPOSE** $m_2)$ (d) $m = (m_1$ **F_COMPOSE** $m_2)$

FIGURE 2 Graphical illustration of the *X_COMPOSE* and *F_COMPOSE* constructs of the specification language. The conventions are those of Figure 1. Note that in the illustration of the *F_COMPOSE* construct the sort or operation *e* is "forgotten".

3.2.1.1 A context free syntax

```
<composed specification>  ::=  <simple specification>
                             | <composed specification>
                               PLUS <simple specification>
                             | <composed specification>
                               COMPOSE <simple specification>
                             | <composed specification>
                               X_COMPOSE <simple specification>
                             | <composed specification>
                               F_COMPOSE <simple specification>
```

The exact definition of a `<simple specification>` may be found in Appendix C, page 96. The difference between a `<simple specification>` and a `<composed specification>` is purely syntactical. It allows to implicitly associate with the compose constructs a lower priority than with other constructs.

3.2.1.2 The context conditions

Let $spec_1$ and $spec_2$ be two specifications. Let $\Sigma_{i_1} = (S_{i_1}, \Omega_{i_1})$, $\Sigma_{i_2} = (S_{i_2}, \Omega_{i_2})$ be the import signatures of $spec_1$ and $spec_2$ respectively and $\Sigma_{e_1} = (S_{e_1}, \Omega_{e_1})$, $\Sigma_{e_2} = (S_{e_2}, \Omega_{e_2})$ their export signatures. Let $C_{i_j}(s)$ be the set of imported constructors of a sort $s \in S_{i_j}$ in $spec_j$ and $C_{e_j}(s)$ be the set of exported constructors of a sort $s \in S_{e_j}$ in $spec_j$, $1 \leq j \leq 2$.

Put

- $im\_op := (\Omega_{i_1} - \Omega_{e_2})$
- $sorts(im\_op) := \{s \mid \ s \ is \ a \ sort \ occurring \ in \ im\_op\}$
- $im\_so := (S_{i_1} - S_{e_2}) \cup sorts(im\_op)$
- $ex\_op := (\Omega_{e_2} - \Omega_{i_1})$
- $sorts(ex\_op) := \{s \mid \ s \ is \ a \ sort \ occurring \ in \ ex\_op\}$
- $ex\_so := (S_{e_2} - S_{i_1}) \cup sorts(ex\_op)$

Let the symbols $\cup$, $\cap$, $-$ denote the union, disjunction and substraction of sets. We now indicate under which conditions the context free rules of Section 3.2.2 may be applied.

For the application of the context free rule with PLUS the following conditions must be met:

- $(S_{e_1} - S_{i_1}) \cap (S_{e_2} \cup S_{i_2})$ is empty.
- $(\Omega_{e_1} - \Omega_{i_1}) \cap (\Omega_{e_2} \cup \Omega_{i_2})$ is empty.
- $(S_{e_2} - S_{i_2}) \cap (S_{e_1} \cup S_{i_1})$ is empty.
- $(\Omega_{e_2} - \Omega_{i_2}) \cap (\Omega_{e_1} \cup \Omega_{i_1})$ is empty.
- all operations of $spec_1$ and $spec_2$ have to be compatible in the following sense: there may not be the same operation name with the same source and target sorts in $spec_1$ and $spec_2$. There may also be no infix operation name in one specification which appears as a component name of a mixfix operation name in the other specification.
- the set of all constructors of a sort s of $(S_{i_1} \cap S_{i_2})$ has to be the same in both specifications.

For an application of the context free rule with COMPOSE the following conditions must be met:

- $S_{i_1} = S_{e_2}$ and $\Omega_{i_1} = \Omega_{e_2}$.
- $(S_{e_1} - S_{i_1}) \cap S_{i_2}$ is empty.
- $(\Omega_{e_1} - \Omega_{i_1}) \cap \Omega_{i_2}$ is empty.
- $C_{i_1}(s) = C_{e_2}(s)$ or $C_{i_1}(s) = C_{i_2}(s)$ or $C_{i_1}(s)$ is a subset of Ω_{i_2} and there is no imported constructor in $spec_2$ with target sort s — for all sorts s which are target sort of an imported constructor of $spec_1$.

X_COMPOSE is a macro defined as follows:

```
spec1  X_COMPOSE spec2
 stands for:

( (spec1  PLUS
                    ( IMPORTS SORTS  ex_so
                              OPNS   ex_op) )
    COMPOSE

      (( IMPORTS SORTS im_so
                 OPNS  im_op)
         PLUS spec2))
```

The context conditions for the context free rule with *X_COMPOSE* result from those of the constituent components.

F_COMPOSE is a macro defined as follows:

spec$_1$ **F_COMPOSE** *spec*$_2$
stands for:

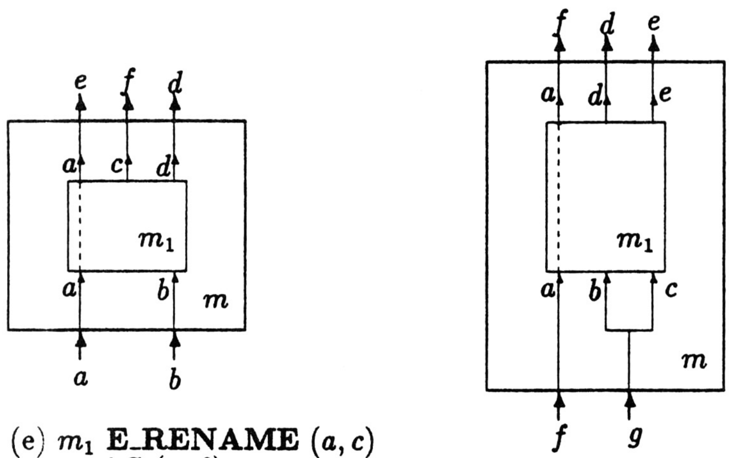
```
( spec1 COMPOSE
    ( (spec2
        FORGET SORTS (Se2 - Si1)
               OPNS ex_op)
      PLUS
        ( IMPORTS SORTS im_so
                  OPNS im_op)))
```

The context conditions for the context free rule with *F_COMPOSE* result from those of the constituent components.

3.2.2 The renaming constructs

There are two renaming constructs renaming exported sorts and operations and imported sorts and operations respectively. The corresponding keywords are *E_RENAME* and *I_RENAME*. The constructs are equivalent to the constructs $[lso1/lso2]m_1$ and $m_1[lso1/lso2]$ of [LL 88].

Figure 3 gives a graphical illustration of these constructs:



(e) m_1 **E_RENAME** (a, c) **AS** (e, f)

(f) m_1 **I_RENAME** (a, b, c) **AS** (f, g, g)

FIGURE 3 Graphical illustration of the renaming constructs of the specification language. The conventions are those of Figure 1.

The corresponding context free rules are

```
<simple specification>  ::=  <simple specification>
                              E_RENAME <rename list>
                            | <simple specification>
                              I_RENAME <rename list>

<rename list> ::=  SORTS { <sort>, } AS SORTS { <sort>, }
                   [ OPNS { <operation>, }
                     AS OPNS { <operation name>, } ]
                 | AS SORTS { <sort>, } OPNS { <operation name>, }

<operation name>        ::=  <name>
                            | <name> _
                            | _ <name> _
                            | <name> _ <name> { _ <name> }
                            | _ <name> { _ <name> }
```

The context conditions are, among others:

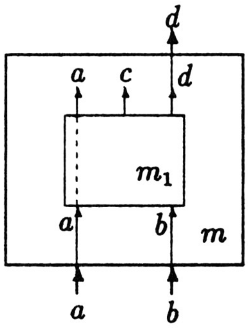
- the number of sorts (operations) on the second list must be the same as on the first list;
- no sort or operation may occur twice on the first list;
- if an operation name on the second list contains underscores, then the operation on the first list corresponding to it must have as many arguments as the operation name on the second list has underscores.

3.2.3 The FORGET construct

The forget construct allows to "forget" exported sorts and operations. Note that "forgetting" a sort implies "forgetting" all operations in which this sort occurs. Note also that the constructors can no longer be exported if one or more constructors are "forgotten" (cf. Sect. 3.1.1).

In addition to the forget construct there is the macro *FORGET_ALL_BUT* which can be used to forget all sorts and operations but a few listed after this keyword.

A graphical illustration of the construct is in Figure 4:



(g) m_1 **FORGET** (a, c)

FIGURE 4 Graphical illustration of the *FORGET* construct of the specification language. The conventions are those of Figure 1.

<simple specification> ::= <simple specification> **FORGET**
<list of sorts and operations>
| <simple specification> **FORGET_ALL_BUT**
<list of sorts and operations>

3.2.4 The axiom constructs

There exist two axiom constructs constraining the imported and the exported algebras respectively. They are characterized by the keywords *I_AXIOMS* and *E_AXIOMS* respectively. Essentially, the imported axiom construct is used to express parameter constraints; the exported axiom construct is used to check properties. A precise semantics of the constructs may be found in [LL 88] and [LL 90], and a discussion of their use in [LL 90].

The axioms are expressed in first-order predicate logic.

The context free rules are:

<simple specification> ::= <simple specification> **E_AXIOMS** <axiom>
ENDAXIOMS
| <simple specification> **I_AXIOMS** <axiom>
ENDAXIOMS

```
<axiom>               ::= <formula>; { <formula>; }
                        | VARS <list of variables>; <formula>;
                          { <formula>; }

<formula>             ::= <disjunction>
                        | <disjunction> { => <formula> }
                        | <disjunction> { <=> <formula> }

<disjunction>         ::= <conjunction> { | <conjunction> }

<conjunction>         ::= <simple formula> { & <simple formula> }

<simple formula>      ::= <equation>
                        | (<formula>)
                        | ! <simple formula>
                        | EX <variable> {, <variable>}
                          .<simple_formula>
                        | ALL <variable> {, <variable>}
                          .<simple_formula>

<equation>            ::= <term> == <term>
                        | <term> [= <term>
```

Note that the symbols **!**, **EX**, **ALL** stand for the logical symbols $\neg$, $\exists$, $\forall$ from classical logic.

3.2.5 The SUBSET and QUOTIENT constructs

The *SUBSET* and *QUOTIENT* constructs build a subalgebra and a quotient algebra respectively. The reader is referred to [LL 88] or [LL 90] for a precise definition.

Context free rules are:

```
<simple specification> ::= <simple specification> SUBSET OF <sort>
                             BY <axiom> ENDSUBSET

<simple specfication> ::= <simple specification> QUOTIENT OF <sort>
                            BY <axiom> ENDQUOTIENT
```

3.3 Modules

Informally, a module is a specification together with its name.

3.3.1 Declaration of a module

A specification is automatically turned into a module when loaded into the module database. The name of the module is the name chosen at the start of the editing of the specification.

3.3.2 Instantiation of a module

A module is instantiated ("called") by its name. The corresponding context free rule is:

`<simple specification>   ::=` **INCLUDE** `<name>`

where `<name>` denotes the module name.

3.3.3 Note

Contrasting with [LL 88] and [LL 90] the OBSCURE system allows no parameterized modules. Actually, a parameter passing mechanism may be simulated by appending an *I_RENAME* construct to the module instantiation: the renamed sorts and operations constitute the formal parameters, the new names the actual parameters (cf. [LL 88], [LL 90]).

Chapter 4

The System OBSCURE

The following sections present a description of the components of the system OBSCURE and indicate how to use them. The last section describes the special editing commands available in *Emacs* to the user.

4.1 Starting and using the system

The following section explains how to start the system and what has to be observed when starting the system for the first time. The information presented in this section is only valid when the system is used together with the *Emacs* editor. This is strongly recommended. If *Emacs* is not available, OBSCURE can still be used by calling the system components from a standard UNIX shell; the synopsis of the calls is presented in Chapter 6.

4.1.1 Starting the system

The system is started by starting the *Emacs* editor. How to do this may be different from system to system (usually, this is obtained by typing `emacs`). Before the system can be used the following remarks have to be observed.

The `obscure-mode` is a special major mode for *Emacs*. Of course, the (code for this) mode has to be loaded into the *Emacs* editor before the mode can be started. The loading of such modes is achieved by the following command of *Emacs*:

`M-x load-file` The user is asked for the name of a file. The file is loaded into the *Emacs* editor and its code is evaluated.

Hence the `obscure-mode` is loaded by typing

`M-x load-file`

and then typing the file name

`obscure.el`

The `obscure-mode` is now available to *Emacs* but not yet started. The following command is used to start the mode:

`M-x obscure-mode`	(also called by: `M-o`)
The *obscure-mode*, a local '`major mode`' in *Emacs*, is started. This means in particular that all special key bindings and variables are available and that the variables are initialized. Moreover all "OBSCURE–commands" are available in the active buffer.	

For a frequent user it is not recommended to follow these steps every time. In Section 4.1.4 some alternatives allowing the automatic loading of the `obscure-mode` are indicated.

4.1.2 Using the system for the first time

When working with the OBSCURE system for the first time, the user should notice the following two points.

(i) After having started the system for the first time, the user will be asked by *Emacs* whether automatic loading of the `VIP-mode` should be suppressed at the start of the `obscure-mode` (the `VIP-mode` is another *Emacs* mode, simulating the behaviour of the well–known *vi* editor). The answer is saved for later sessions in the user's personal '`.obscure`' file. If the user later changes his/her mind and wants the `VIP-mode` to be active during a session, he/she can achieve this by setting the variable `o-vip-desired` to the value `t`. If he/she wants to switch it off again, he/she resets it to the value `nil`.

(ii) As explained in the introduction (see 1 [Introduction], page 1), the OBSCURE system provides module databases. Each user may have a single personal database, in which all his/her specification modules are entered. This module data base has to be created and initialized during the first session with the system. This is performed by the command

`o-mdb-install`

to be given with *Emacs* in `obscure-mode`. (see 4.3 [The module database], page 58 for more information.)

4.1.3 Using the system together with Suntools

If the OBSCURE system is started under Suntools, a so-called *OBSCURE-menu-window* appears above the left edge of the window *Emacs* is running in. This window makes working with the system more comfortable. It contains several buttons, two lines for text input and a line for system messages. The *OBSCURE-menu-window* is connected to an *Emacs* window via pipes. Besides '`The End`' and '`Help`', all functions chosen in the *OBSCURE-menu-window* are executed in the *Emacs* window.

The choice of a button is made by pointing with the mouse pointer at it and then pressing the left mouse button. The pressing of the middle mouse button starts the *Emacs-info-documentation-reader* in the *Emacs* window, which displays the help information available for the function chosen. The pressing of the right mouse button in the *Emacs* window of a buffer, which is in the `obscure-mode`, opens up a submenue offering additional functions.

The following list shows the correspondence between the buttons in the window and the commands of the `obscure-mode`:

- *Button label*: MDB-select
  Name of the command: (`o-mdb-select`)
  Documentation: (See 4.3 [The module database], page 58)
- *Button label*: Interpreter
  Name of the command: (`o-interpreter`)
  Documentation: (See 4.4 [The interpreter], page 66)
- *Button label*: Signature
  Name of the command: (`o-act-signature`)
  Documentation: (See 4.6 [Special editing commands], page 71)
- *Button label*: Parser
  Name of the command: (`o-parser (buffer-name (current-buffer)) o-parser-opts`)

  Documentation: (See 4.2 [The parser], page 57)
- *Button label*: Module Graph
  Name of the command: (`o-module-graph (buffer-name (current-buffer))`)
  Documentation: (See 4.6 [Special editing commands], page 71)
- *Button label*: Load Module
  Name of the command: (`find-file` FILENAME)
  Documentation: (FILENAME is constructed from the values of the text input fields '`Directory`' and '`Modulename`'. These fields also show the chosen, possibly new, directory or filename during the execution of the `find-file` command.)
- *Button label*: Parser Options
  An additional menu window is opened. The options for the parser can be set with the help of this window.
  Documentation: (See 4.2 [The parser], page 57, for more info.)

4.1.4 Advanced procedures

In Section 4.1.1 it was shown how to start the **obscure-mode**. This procedure is too cumbersome for frequent usage. Instead, a frequent user is recommended to add with the help of an editor (*Emacs* or *vi*, for example) *one* of the following two function calls to his/her personal '**.emacs**' file:

```
(autoload 'obscure-mode "obscure.elc" "" t)
(setq auto-mode-alist (cons '("
.T$" . obscure-mode) auto-mode-alist))
```

or

```
(load "obscure.elc")
```

The difference between these two possibilities is as follows.

The former possibility turns the function **obscure-mode** into a user command of *Emacs* and states that the code of the function is to be found in a file called **obscure.elc**. *Emacs* will also start the **obscure-mode** automatically whenever a file ending with '**.T**' is loaded into *Emacs*.

The latter function call causes the automatic loading of the file '**obscure.elc**' containing the **obscure-mode** at the start of *Emacs*. In that case the **obscure-mode** has then still to be started as explained in Section 4.1.1.

If the user wants a given function to be executed at each start of the **obscure-mode**, he/she has to add the following function call to his '**.emacs**' file:

```
(setq obscure-mode-hook '<function to be called or its definition>)
```

4.2 The parser

The parser checks a specification from an *Emacs* buffer or a file with respect to the context free syntax and the context conditions described in Chapter 3. If a specification is syntactically correct, the parser creates an internal representation for it and writes it into the personal module database.

The user should note the following points:

- the name of the specification to be parsed (i.e. the name of the buffer or file containing the text of the specification) must end with '`.T`';
- the name of an OBSCURE specification in the module database is the same as the name of the file containing its text, but ends with '`.O`'.

The different commands related to the parser are explained in the following.

`M-x o-parser` (also called by: `C-o p` or `C-o C-p`)

The user is asked for a *filename* and *options*.

The file *filename* (note that the filename completion mechanism of *Emacs* can be used when entering a filename such as *filename*) is parsed according to the options *options*. The default value of *filename* is the name of the file associated with the active buffer (See Section 1.2, p. 3 for more information) if it exists. Error messages and other relevant messages are displayed in a special buffer called **`*compilation*`**.

The options can be entered with or without a '-' sign. The following options are possible:

-e The text on which the parser is working is displayed during the parsing.

-f The standard output is directed to '**`FILENAME.E`**' (instead of the buffer **`*compilation*`**).

-v The version number of the OBSCURE parser is displayed.

It is possible to give no option.

When a file is parsed correctly an internal representation of it is entered into the user's personal module database. In the case an entry with the same name already exists, the user is asked whether the entry shall be overwritten. If overwriting is not desired, the specification is parsed but the internal representation is not entered into the module database.

If the specification to be parsed contains an instantiation (*INCLUDE*) of another specification module the text of which has been modified since its last parsing, an error message is displayed and the parser is stopped.

After a call of the parser the command **`next-error`** (`C-x '`) is available as an *Emacs* command and can be used to jump to the lines of the specification containing an error.

`M-x next-error` (also called by: `` C-x ` ``)

The buffer containing the latest specification parsed becomes the active buffer, the cursor is set to the beginning of the line containing the next error and the buffer `*compilation*` is scrolled, so that the next error message is in the top line.

The following variable controls the working of the OBSCURE parser. Its value can be modified by the user.(See 1.5 [Some usefull Emacs Commands], page 5, for more information.)

`o-parser-opts`

This variable contains the default value of *options*, that is the value of *options* at the last call of the parser.

The sorts `bool` and `integer` together with the corresponding constructors and operations are known to the parser by default. They may be omitted from the lists of the imported sorts and operations of a specification. (See C [The context free syntax for the specification language], page 96, for more info.)

4.3 The module databases

This section explains the structure of the module databases of the system and introduces the corresponding query language.

4.3.1 General concepts

A database consists of a collection of data in a structured form. The data in a database are ordered in *tables*. A table contains only data with the same "structure".

Each table has a name and consists of *lines* and *columns*. Data items in the same column are supposed to refer to the same "concept"; data items on the same line are supposed to refer to the same "object". For instance, a column may contain (names of) persons and a line the name, age, sex, etc. of a person. A column is called a *field*, a line is called a *(data) record*. Each field has a name called *field name* or *attribute*.

4.3.2 The structure of the module databases

The following sections describe the structure of a module database and the user interface to these module databases.

The module databases of the OBSCURE system contain information about the specifications built with the system.

A distinction is made between *personal* module databases and the *global* module database. Each user of the system can install his/her personal module database (see Section 4.3.3, `o-mdb-install`). The global module database contains specifications that are of interest to any user; only a user with the account *mdbpool* has a writing access to it.

Each user has reading access to all module data bases but a writing access to his personal module database only. More precisely, he/she can enter a specification only into his/her personal database through the parser. The requirement to use the parser makes sure that a database contains syntactically correct specifications only. Moreover, the user can delete a specification from his/her personal module database only.

The names

The global module database is called '`mdbpool`'.

The user installs his/her personal module database with the command `o-mdb-install` as explained in Section 4.3.3. The name of the module database thus created is the same as the account of the user calling `o-mdb-install`.

The attributes

A module database contains the following attributes:

- the name of the specification;
- the imported sorts of the specification;
- the exported sorts of the specification;
- the imported operations of the specification;
- the exported operations of the specification;
- the imported constructors of the specification;
- the exported constructors of the specification;
- the import axioms of the specification;
- the export axioms of the specification;
- the list of the specifications, that are instantiated by this specification (uses-entry);
- the list of the specifications in which this specification is instantiated (is-used-by-entry);
- an attribute showing whether the specification has to be parsed again.

The name of a specification constitutes a key attribute because it univocally characterizes a data record of the database.

Queries

A query is performed by the command `o-mdb-select` to be described in Section 4.3.3. Essentially, a query is characterized by three *query parameters*:

- a database name;
- a list of attribute names;
- a condition.

The answer to a query consists of a list of attributes. More precisely, the attributes are those identified by the second query parameter; they belong to the module(s) satisfying the third query parameter; these modules belong to the data base identified by the first query parameter.

4.3.3 The commands

The command `o-mdb-install`

`M-x o-mdb-install`

> An empty module database is created, if there exists not yet a personal module database for the user. If such a personal module database already exists nothing happens.
>
> The name of the module database created is the same as the account of the user calling `o-mdb-install`. When the module database has been created the message '`Database created`' is displayed in the minibuffer.

The command `o-mdb-select`

This command performs a query.

M-x o-mdb-select (also called by: **C-o s** or **C-o C-s**)

A frame for the query parameters containing the values of the last call of `o-mdb-select` is displayed in the buffer **`*Data-base-input*`**. This buffer may for instance look as follows:

```
*************************** Select from MDB *********************
 C-i start the search in the data base
 C-c start selecting field names  ESC escape from selecting field
                                                           names
 RET select a field name   C-t toggle all fields
 SPC move to the next field name   DEL move to the previous field
                                                           name
*******************************************************************

 SELECT FROM
 Data base:  arbor
  List of field names:  mname
  iparams [ ] eparams [ ] isorts [X] esorts [X]
  iopns [X] eopns [ ] iconstr [ ] econstr [X]
  iaxioms [ ] eaxioms [ ] uses [ ] isusedby [ ] compiled [X]
WHERE Condition:  mname LIKE "int*"
```

The field names *iparams* and *eparams* are—at the present stage of development of the system—irrelevant; the field names *isorts*, *esorts*,...,*compiled* are the attributes mentioned in Section 4.3.2.

The first query parameter is the database name following "**`Data base:`**". The second query parameter is the list of attribute names identified by the **`[X]`**. The third query parameter, viz. a condition, is consists of the text following "**`WHERE Condition:`**".

The user can now modify the query parameters by using the commands displayed in the top four lines of the frame. More precisely, he may add or delete "`X`" between the bracket pairs `[ ]` and he may modify the database name and the condition, i.e. the text following "**`Database:`**" and "**`WHERE Condition:`**". The query itself is then started by **`C-i`** and consists in searching in the module database for those modules that satisfy the condition and in displaying their attributes.

The syntax and semantics of the *condition* constituting the third query parameter can be found in Section 4.3.5, page 63.

Examples of database queries can be found in the chapter **Examples** (see 5.2 [Example queries ...], page 85).

The command `o-mdb-delete`

This command deletes a data record, i.e. a module, from the module database.

`M-x o-mdb-delete` (also called by: `C-o d` or `C-o C-d`)

> The user is first asked by *Emacs* for a *specification name* and a *database name*. The data record corresponding to the specification *specification name* is then deleted from the module database *data base name* (default-value: name of the personal module database). The file with the internal representation of the module is also deleted. The user is asked, whether he wants a protocol of the deletions. Messages of success or errors and possibly the protocol are displayed in the buffer `*MDB-OUTPUT*`.
>
> A data record and its internal representation can only be deleted if the following condition is fulfilled:
>
> - there is no entry for the attribute `ISUSEDBY` of the specified data record.
>
>   This condition avoids deleting a specification used by another specification.

The command `o-mdb-help`

`M-x o-mdb-help`

> Starts the *Emacs* information reader with the description of the module database.

4.3.4 The variables

The following variables, the value of which the user can modify (see 1.5 [Usefull Emacs commands], page 5 for more information), control the behaviour of the commands accessing a module database:

`o-mdb-out`

> This variable indicates where the answers to interactions with the module database (`o-mdb-select`, `o-mdb-delete`) may be found. The default value of this variable is '`STDOUT`', which means that the output is displayed in the buffer `*MDB-OUTPUT*`. If the output is to be written into a given file, the variable `o-mdb-out` must contain the name of that file.

`o-lfieldname`

This variable contains the list of field names of the module database chosen at the last call of `o-mdb-select`. This list constitutes the second query parameter.

`o-condition`

This variable contains the third query parameter of the last call to `o-mdb-select`.

`o-mdb`

This variable contains the name of a module database, i.e. the first query parameter of the last call of `o-mdb-select`. The default value is the name of the personal database.

4.3.5 The query parameter <condition>

The now following syntax and semantics of the third query parameter may seem too sophisticated for a system like OBSCURE. In practice, only a small fraction of the facilities offered by this query parameter will effectively be used. The reason for the high degree of sophistication lies in the fact that the database system was originally developed for another project.

4.3.5.1 The syntax

Note that the syntax now following differs from the syntax of the specification language OBSCURE.

The grammar is given in EBNF.

```
<condition>          ::= <table field> <cmpop> <table field>
                       | <table field> <cmpop> <value>
                       | <table field> LIKE <expression>
                       | <table field> UNLIKE <expression>
                       | <string constant> IN <table field2>
                       | <condition> AND <condition>
                       | <condition> OR <condition>
                       | NOT <condition>
                       | ( <condition> )
```

```
<table field>              ::= <attribute name>
                             | <name>.<attribute name>

<table field2>             ::= isorts | esorts | uses
                             | isusedby | iopns | eopns
                             | iconstr | econstr
                             | iaxioms | eaxioms

<value>                    ::= <string constant>

<string constant>          ::= “ sequence of ASCII symbols ”

<expression>               ::= “ regular expression ”, explained in 4.3.4.3

<cmpop>                    ::=  = | <= | >= | < | > | !=

<attribute name>           ::= mname | isorts | esorts
                             | iopns | eopns | iconstr
                             | econstr | iaxioms | eaxioms
                             | uses | isusedby | compiled
<name>                     ::= <letter> { <alphanumerical symbol> }

<alphanumerical symbol>    ::= <letter> | <digit>

<letter>                   ::= A | B | C ...| a | b | c ... | z

<digit>                    ::= 0 | 1 | 2 | ... | 9
```

For examples on the usage of the conditions see 5.2 [Example queries ...], page 85.

4.3.5.2 The semantics

The semantics of the conditions is as one would expect and as suggested by the names of the predicates (**AND**, **OR**, ...). The usual priorities hold for the operations NOT, AND and OR, i.e. NOT binds strongest, then AND follows. OR has lowest priority.

The relations defined by `<cmpop>` implement the usual lexicographic order on strings with respect to the order on the ASCII symbols.

LIKE and **UNLIKE** are used to search for entries into the given table field that match the given regular expression. All matching entries are outputted. **IN** checks whether the string on its left hand side occurs in the table field identified by its right hand side.

4.3.5.3 The regular expressions

As indicated above the `<expression>` following **LIKE** and **UNLIKE** is a regular expression. This expression has to be of the following form:

Let "ASCII" be the set of all ASCII symbols. Let x,y,z be symbols from "ASCII".

A regular expression is	Meaning
x	the symbol x
?	any symbol from "ASCII"
[xyz]	the symbol is either x or y or z
[^xyz]	the symbol is neither x nor y nor z
[x-y]	the symbol is in the range x to y
[^x-y]	the symbol is not in the range x to y
*	string of any length (including the empty string

Additional rules:

1. A symbol of the set of special symbols { \, “, ”, ?, [,], * } must not appear between [and]. A symbol of the set of symbols { \ , [,], -, ^ } may appear between [and]. If a special symbol shall be represented as a symbol of the alphabet “ASCII”, then it has to be preceded by “\”.
2. The symbol “^” for the negation may appear only directly after the opening bracket between [and]. The symbol “-” for ranges has to be between two symbols of the alphabet “ASCII”. Furthermore, successions of the form [x-y-z] are not allowed. No other special symbols may appear between the brackets [ ].
3. Between the pair of symbols [ ], there has to be at least one symbol of the alphabet “ASCII”.
4. The special symbol “*” may not be followed directly by another “*”.

Some examples of possible regular expressions are:

- `N?TSTACK`—matches all strings starting with a `N` followed by an ASCII symbol and ending in `TSTACK`;
- `NAT*STACK`—matches all strings starting with `NAT` and ending in `STACK`;
- `N[aeiou]TSTACK`—matches all strings starting with a `N` followed by a vowel and ending in `TSTACK`;
- `NA[^aeiou]STACK`–matches all strings starting with `NA` followed by a character which is not a vowel and ending in `STACK`.

4.4 The interpreter

The goal of the OBSCURE interpreter is the evaluation of terms.

The interpreter essentially reduces input terms to output terms. The notions of input and output terms are defined in the following:

Any term over a current specification importing basic sorts and operations only (see C [The context free syntax for the specification language], page 96 for more information.) is an *input term*. The term may contain *bound* variables which have to be identifiers starting with the symbol **$**.

Output terms are the result of the evaluation of input terms and consist of constructors only.

As already indicated in Chapter 3 it is possible to specify infinite datatypes (e.g. infinite lists). This leads to termination problems when interpreting terms. The interpreter therefore works on the basis of '`call-by-need`' semantics: the values of variables (that is formal parameters and variables that develop from the bindings in a *CASE*-term) are computed only if they are accessed. More precisely, the specifier has the possibility to delay the evaluation of arguments to constructors by marking the constructors in the specification with the keyword *LAZY* ("lazy specification"). More information on the subject may be found in [Sto 91]. For an example of a "lazy specification" and of *lazy-evaluation* with the interpreter see Section 5.1.2.2, page 78.

4.4.1 The commands

The interpreter is embedded in a so-called *interpreter mode*. The call of this mode is explained in the following. The special commands of the mode are explained in Section 4.4.2.

`M-x o-interprete`	(also called by: `C-o i` or `C-o C-i`)
This command starts the *interpreter mode*. The communication between the user and the interpreter happens via the *Emacs* buffer **`*Interpreter*`**. It contains the *interpreter mode* and all interaction with the interpreter happens through this buffer.	

4.4.2 The interpreter mode

The *interpreter mode* (called OIM for short) knows the following commands:

- **currspec** *name*
  The specification *name* becomes the current specification provided it exists in the module database, it is marked as compiled and it does not import sorts or operations other than basic ones (see Appendix C, p.97). If the specification exists, but is marked as uncompiled, a list of specification names which are contained in this specification is displayed as well as all specifications used by the specifications marked as uncompiled. If '**`currspec`**' is called without *name* the name of the current specification is displayed.

- **let** *name*=*term*
  This command is used to declare a variable and its binding. The text of *term* is textually bound to the string *name*. *Name* must be a string starting with '**`'$'`**' and not containing " ", "\n", "\t" or "=".

- **show** *name*
  Displays the binding of the variable *name*. The conventions for *name* are as for the **let** command.

- **show**
  Displays all variables with their bindings.

- **delete** *name*
  Deletes variable *name*.

- **delete**
  Deletes all variables.

- **write** *name*
  Saves all variables and bindings in the file '**`name.B`**'.

- **read** *name*
  Reads variables and bindings from the file '**`name.B`**'. Any bindings of variables that also occur in the '**`name.B`**' are overwritten.

- **eval** *term*
  Replaces all names in *term* starting with '**`$`**' by their corresponding value, parses the term and evaluates it with respect to the current specification. If some names starting

with '**$**' have no binding, a list of those names is displayed. A current specification must be given (via the '**currspec**' command) before the first **eval** command.

- **varbind yes**
  Only the bindings of the variables occurring in constructor terms in argument places which are declared as *LAZY* in the constructor list of the specification are displayed (in square brackets).

- **varbind no**
  All bindings are displayed.

- **trace yes**
  The '`trace mechanism`' is switched on. The next call of the **eval** command starts the *trace mode*. The *trace mode* has its own prompt: *(Otm)*. The following commands are available in the *trace mode*:

  1. **help**
     Starts the *Emacs-information-reader* with the description of the interpreter.
  2. **break** *term*
     Sets up a '`breakpoint`' in the form of a term. This *term* must be an input term over the signature of the current specification. Only the topmost function symbol of *Term* is relevant, i.e. if the breakpoint is *f(x,y)* and the term to be evaluated next is *f(g(x,y),z)* then the evaluation is stopped, because the topmost function symbols are the same.

     A number is associated with each breakpoint, i.e. the first breakpoint set up by the user gets the number 1 and so on.
  3. **list**
     Lists the breakpoints in the order they have been set up.
  4. **delete** *integer*
     Deletes the breakpoint with the number *integer*.
  5. **run**
     Resumes the evaluation until the next breakpoint is reached.
  6. **step**
     The next step of the evaluation is started.
  7. **next**
     The subterm to be evaluated next is evaluated without displaying the intermediate steps of the evaluation. For instance the original term has been *f(g(x,y),z)* and the term to be evaluated next is *g(x,y)*, then *g(x,y)* is evaluated without displaying the intermediate steps of the evaluation. If a breakpoint is reached during this evaluation the evaluation is stopped.
  8. **modify** *term*
     If a term that is a variable is to be evaluated next its binding can be changed via this command into the given input term *term* of the current specification. If the command is entered at a point where it can not be executed (no evaluation of a term that is a variable comes next), an error message is displayed.

9. **where**
   This command displays the "hierarchy" of functions to be evaluated. For instance if the subterm $g(h(x))$ of term $f(g(h(x)))$ is to be evaluated next, the command would display $g(h(x))$ and then $f(g(h(x)))$.

When the evaluation is finished the system is still in *trace mode*. The following command named **trace no** can be used to leave the *trace mode*:

- **trace no**
  The '`trace mechanism`' is switched off.

- **quit**
  Quits the *interpreter mode* and closes the connected window.

- **help**
  Starts the *Emacs-information-reader* with the description of the interpreter given above.

It is sufficient to enter unambigous prefixes of the commands in the *interpreter mode* and the *trace mode*, i.e. one may write `c` instead of **currspec**.

Quitting the interpreter, the hard way

The user can leave the *interpreter mode* at any time—also from within the *trace mode*—by pressing `C-c` twice. The message "really leave OIU (y/n)?" is displayed and the *interpreter mode* is quitted if 'y' is pressed. The *Emacs* window containing the *interpreter mode* disappears from the screen.

Changing the prompt of the interpreter

The prompt of the interpreter mode: ";-) " is used by default. It can be modified by changing the value of the shell variable OIU_PROMPT. This has of course to be done in the standard UNIX shell.

For example: `setenv OIU_PROMPT '><:'`

For examples illustrating the use of the interpreter and the trace mechanism the reader is referred to Section 5.1.2 and 5.1.2.3 respectively.

4.5 The Source-to-Source-Translation of specifications

`M-x o-So-to-So-Tra` (also called by: `C-o u` or `C-o C-u`)

After starting this command the user is asked for the name of a specification *spec* and for an option *opt*.

The specification *spec*, which must be contained in the personal module database of the user, is translated into C. The default value for *spec* is the name of the active buffer.

The option *opt* defines the target language. As in its present state the system contains a compiler into C only, so the only option possible is '`c`'.

At the present moment only atomic specifications can be translated. If the specification is not atomic an error message is displayed. After the translation, the source code created is displayed in a corresponding *Emacs* buffer; more precisely after the call of `o-So-to-So-Tra` with the parameters *spec* and '`c`' the files '`spec.h`' and '`spec.c`' are displayed in two corresponding *Emacs* windows.

The source code created by `o-So-to-So-Tra` can in principle be included into a program written in C. In that case the following syntactical rules have to be observed in order to match the syntax of C:

- the names of operations and variables must conform to the C-syntax;
- the symbol prime may not occur in variable names.

Note that the mixfix operation names occurring in a specification are concatenated into a single name after translation.

4.6 Special editing commands

The following commands support the editing of specifications by saving some writing.

`M-x o-curr-signature` (also called by: `C-o t` or `C-o C-t`)

The current signature, i.e. all imported and exported sorts and operations of the specification in the active buffer, is displayed in an *Emacs* buffer called `*Signature*`. If the specification has not yet been parsed, a corresponding error message is displayed.

`M-x o-at-spec` (also called by: `C-o a` or `C-o C-a`)

The contents of the buffer, the name of which is (interactively) entered by the user, is displayed in the active buffer. Moreover, the frame of a specification is inserted into the active buffer, if the user confirms a corresponding query. If the buffer does not exist, a new buffer is created, and the user is asked into which file its contents should be saved. Instead of a filename the user can also enter `C-g`. In that case no file is connected to the buffer.

`M-x o-copy-opns` (also called by: `C-o c` or `C-o C-c`)

The list of operations following the keyword *CREATE* is copied into the active buffer behind the keyword *CONSTRS*.

`M-x o-copy-constrs` (also called by: `C-o z` or `C-o C-z`)

The list of constructors following the keyword *CONSTRS* is copied onto the top of the list of operations following the keyword *CREATE*.

`M-x o-ins-comment` (also called by: `C-o k` or `C-o C-k`)

If the current line is not a comment line already it is marked as such by inserting '`##`' at the very left of the line.

`M-x o-kill-comment` (also called by: `C-o K` or `C-o C-K`)

The comment markers '`##`' are deleted from the current line.

`M-x o-print-spec` (also called by: `C-o w` or `C-o C-w`)

The user is asked for the name of a specification *spec* and the name of a module database *mdb*. A textual representation of the specification module *spec* contained in the module database *mdb* is written into the *Emacs* buffer `*Specification*`. The module name of the last call of `o-print-spec` is the default value of *spec*. The following options control the behaviour of the command:

-m: prints the specification while expanding the three macros *X_COMPOSE*, *F_COMPOSE* and *FORGET_ALL_ BUT*.

-r: prints the specification while replacing each *INCLUDE* construct by the specification it stands for;

-b*number*: sets the linewidth for the output to *number*; the default value is 75.

-v: prints the version number of the printer. This can be used to control whether the newest version of the printer is installed. The number of the current version is 1.2.

`M-x o-news` (also called by: `C-o n` or `C-o C-n`)

The file '`OBS-NEWS`' is loaded and displayed if it has changed since its last reading. Otherwise, the message '`No News`' is displayed.

`M-x o-show-constrs` (also called by: `C-o h` or `C-o C-h`)

The user is asked for the name of a sort. If the active buffer contains an atomic specification the list of constructors of this sort is displayed in the *Emacs* buffer `*Constructors*`.

`M-x o-opns-to-def` (also called by: `C-o o` or `C-o C-o`)

If the active buffer contains an atomic specification then the list of all operations to be defined by recursive programs is displayed in the *Emacs* buffer `*Opns-to-be-defined*`.

`M-x o-LaTeX-format` (also called by: `C-o l` or `C-o C-l`)

The user is asked for the name of a file which should contain the text of a specification. The contents of this file are translated into LaTeX format according to the information in the variables *o-sym-file*, *o-word-file* and *o-other-opts*. If the name of the file is *spec*.T then the LaTeX format is displayed in the buffer *spec*.tex. The following options are possible:

-h: displays help information;

-i*infile*: the text of the specification is to be read from the file *infile*; the default value is '`stdin`';

-o*outfile*: the translation is written into the file *outfile*; the default value is '`stdout`';

-t*number*: defines the width of a tabulator stop; the default value is 8;

-n*[l—r][e][number]*: switches the line numbering on; the default value is '`off`'. The optional arguments *l-r*, *e* and *number* indicate whether the number has to be printed to the right or to the left, whether empty lines should be numbered, and which is the starting number;

-x: if this option is set, the usual convention of LaTeX ignoring several successive blanks is circumvented; this means that all blanks in the input become blanks in the output too;

-v*length*: defines the distance to be added for empty lines. The default value is 3mm;

-s*symfile*: the format information is to be read from the file *symfile*;

-w*wordfile*: the reserved words (to be printed in bold face) are to be read from the file *wordfile*.

Note that this command is rather powerful and can be used to format texts in any programming language by modifying *symfile* or *wordfile* according to the language in question.

Chapter 5

Examples

The following sections give examples of specifications, of queries to the module database and of evaluations by the interpreter.

5.1 Examples of specifications

In the following section a simple example for a specification is given. It is the example already treated in Chapter 2. For a more complex example see Appendix D.

5.1.1 Simple examples

Three specifications are presented: a specification of the natural numbers, a specification of stacks (of elements) and a specification of stacks of natural numbers.

In the OBSCURE system each module has to be saved into a file with a name, of the form '`<name of the module>.T`'. In the now following text the name of the module is written on the first line as a comment.

(i) A specification of natural numbers

```
## nat.T
 CREATE
  SORTS nat
  OPNS null: -> nat
         succ: nat -> nat
         add:  nat nat -> nat
 SEMANTICS
  CONSTRS null: -> nat
            succ: nat -> nat
  VARS
  i1, i1', i2: nat
  PROGRAMS
  add (i1, i2) <-
    CASE i1  OF null: i2;
                succ(i1'): succ(add(i1', i2))
    ESAC;
  ENDCREATE
 ## The End
```

(ii) A specification of a stack of elements

```
## element-stack.T
 IMPORTS
  SORTS element
 CREATE
  SORTS stack
  OPNS  empty: -> stack,
         push: element, stack -> stack
         pop: stack -> stack
         top: stack -> element
 SEMANTICS
  CONSTRS
     empty: -> stack,
     push: element, stack -> stack
  VARS s,s': stack,
      e: element
  PROGRAMS
  pop(s) <-
    CASE s  OF
       empty:  ERROR(stack);
       push(e, s'): s'
    ESAC;

  top(s) <-
    CASE s  OF
       empty:  ERROR(element);
```

```
      push(e, s'): e
    ESAC;
  ENDCREATE
 ## The End
```

(iii) A specification of a stack of natural numbers

```
## nat_stack.T
( INCLUDE element-stack
  I_RENAME
   SORTS element
  AS
   SORTS nat)

 COMPOSE

( INCLUDE nat)
## The End
```

5.1.2 Examples illustrating the use of the interpreter

5.1.2.1 A first example

Let NAT be a specification of the natural numbers with the constructors

```
succ: nat -> nat and
null: -> nat
```

and the function `add: nat nat -> nat` defined by:

```
add (i1,i2) <-
 CASE i1 OF
  null     : i2 ;
  succ(i1'): succ(add(i1',i2))
 ESAC ;
```

The following protocol illustrates the use of the interpreter (remember that `;-)` is the prompt of the *interpreter mode*):

```
;-) currspec NAT

;-) eval add(succ(null),succ(null))

 succ(succ(null))

;-) eval succ(succ(null))

 succ(succ(null))

;-) eval add(i1,i2)

Error message of the term parser:
  /local_pathname/mdbpool/username/NAT.O, line 1: token: "i1"; error:
  here the name "i1" is declared neither as prefixname nor as infixname
  nor as mixfixname nor as variablename
  /local_pathname/mdbpool/username/NAT.O, line 1: token: "i1"; error:
  syntax error
  error
```

The reason for the error message lies in the fact that i1 and i2 are free variables.

```
;-) eval add(add(null,null),null)

 null
```

```
;-) eval succ(add(null,succ(null))

 succ(succ(null))
```

5.1.2.2 Example illustrating lazy evaluation

The following specification specifies infinite and finite lists of natural numbers. Note the use of *LAZY*:

```
## all_nat_list.T
CREATE
 SORTS inf_natlist, fin_natlist

 OPNS
  lcons : integer, inf_natlist -> inf_natlist
  cons : integer, fin_natlist -> fin_natlist
  nil : -> fin_natlist
  all_ints : integer -> inf_natlist
  show1 : inf_natlist, integer, inf_natlist -> inf_natlist
  show2 : inf_natlist, integer, fin_natlist -> fin_natlist

SEMANTICS
 CONSTRS
  lcons : integer, LAZY inf_natlist -> inf_natlist
  ## (Remark: thanks to LAZY the second argument of lcons
  ## is evaluated only when required!)
  cons : integer, fin_natlist -> fin_natlist
  nil : -> fin_natlist

 VARS t , j : integer
      il , il' , init_inf : inf_natlist
      init_fin : fin_natlist

PROGRAMS

 all_ints (j) <-
  lcons (j,all_ints (j+1)) ;
## Remark: Had the second argument of the constructor lcons not been
##         declared as LAZY, the function all_ints would have been
##         undefined for any argument.

 show1 (il , t , init_inf) <-
  IF t >= 1
  THEN CASE il OF
        lcons (j,il') : show1 (il' , t - 1 , lcons (j,init_inf) ) ;
       ESAC
```

```
    ELSE init_inf
    FI;
  ## show1 gets the first t elements from a list il and puts them in
  ## reverse order into the argument init_inf.
  ## init_inf is of sort inf_natlist.
   show2 (il , t , init_fin) <-
    IF t >= 1
    THEN CASE il OF
          lcons (j,il') : show2 (il' , t - 1 , cons (j,init_fin) ) ;
         ESAC
    ELSE init_fin
    FI;
  ## the same as show1, but the argument init_fin is of sort fin_natlist.

  ENDCREATE
```

The following protocol illustrates the use of the interpreter:

```
;-) currspec all_nat_list

;-) varbind yes

## Only the bindings of variables occurring in LAZY arguments of
## constructors are displayed (in square brackets).

;-) eval all_ints(1)

 lcons(1,all_ints(j[1]+1))

;-) eval show1(all_ints(1),0,ERROR(inf_natlist))

 ERROR

;-) eval show1(all_ints(1),2,ERROR(inf_natlist))

 lcons(2,init_inf[lcons(j[1],init_inf[ERROR(inf_natlist)])])

;-) eval show2(all_ints(1),2,nil)

 cons(2,cons(1,nil))
```

5.1.2.3 Example illustrating the trace mechanism

Consider the following specification of natural numbers together with a sort *sort* consisting of a single carrier:

```
## integer.T
CREATE
 SORTS int,sort
 OPNS null: -> int
      succ: int -> int
      ander : -> sort
      add:   int int -> int
SEMANTICS
 CONSTRS null: -> int
         succ: int -> int
         ander : -> sort
VARS
 i1, i1', Arbeit, i2, i2' : int
PROGRAMS
 add (i1, i2) <-
  CASE i2 OF null: i1;
             succ(i2'): succ(add(i1, i2'))
  ESAC;
ENDCREATE
```

The specification shown above is loaded into the interpreter with the command '**currspec**'.

```
;-) currspec integer
;-) trace yes
;-) eval add(null,add(null,null))
You are in the OBSCURE trace mechanism on term level
*********************************************************
You want to interpret the following term :
add(null, add(null, null))
*********************************************************
the next term, which will be interpreted :
add(null, add(null, null))
 (Otm) step
you are in the CASE-term with evaluation Nr. : 1
CASE i2
OF
null:
i1;
succ(i2'):
succ(add(i1, i2'));
ESAC
this CASE-term is enclosed by the term :
add(null, add(null, null))
 (Otm) step
the evaluation of the input term of the CASE-term
with the evaluation Nr. : 1 begins !
the next term, which will be interpreted :
i2[add(null, null)]
```

```
 (Otm) step
the next term, which will be interpreted :
add(null,
null)
 (Otm) step
you are in the CASE-term with evaluation Nr. : 2
CASE i2
OF
null:
i1;
succ(i2'):
succ(add(i1, i2'));
ESAC
this CASE-term is enclosed by the term :
add(null, null)
 (Otm) where
add(null, null)
is called in :
add(null, add(null, null))
 (Otm) step
the evaluation of the input term of the CASE-term
with the evaluation Nr. : 2 begins !
the next term, which will be interpreted :
i2[null]
 (Otm) step
the next term, which will be interpreteted :
null
 (Otm) step
you are again in the CASE-term
 with evaluation Nr. : 2
CASE i2
OF
null:
i1;
succ(i2'):
succ(add(i1, i2'));
ESAC
this CASE-term is enclosed by the term :
add(null, null)
 (Otm) step
the next term, which will be interpreted :
i1[null]
 (Otm) next
the next term, which will be interpreteted :
null
 (Otm) next
the evaluation of the term:
null
```

```
has been finished and delivers ->
null
do acknowledge !
(press enter key, if you want to go on)
the evaluation of :
add(null, null)
has been finished and delivers ->
null
do acknowledge !
(press enter key, if you want to go on)
you are again in the CASE-term
 with evaluation Nr. : 1
CASE i2
OF
null:
i1;
succ(i2'):
succ(add(i1, i2'));
ESAC
this CASE-term is enclosed by the term :
add(null, add(null, null))
 (Otm) step
the next term, which will be interpreted :
i1[null]
 (Otm) modify ander
sort-conflict : sort of your term and sort of the variable does not
agree !
 (Otm) modify succ(null)
variable modified !
the next term, which will be interpreted :
i1[succ(null)]
 (Otm) next
the next term, which will be interpreteted :
succ(null)
 (Otm) next
the evaluation of the term:
succ(null)
has been finished and delivers ->
succ(null)
do acknowledge !
(press enter key, if you want to go on)
the evaluation of :
add(null, add(null, null))
has been finished and delivers ->
succ(null)
do acknowledge !
(press enter key, if you want to go on)
The term reduction has finished and delivers the result :
```

```
succ(null)
You are leaving the OBSCURE trace mechanism !

;-)  currspec integer
;-)  trace yes
;-) e add(null,null)
You are in the OBSCURE trace mechanism on term level **
**********************************************************
You want to interpret the following term :
add(null, null)
**********************************************************
the next term, which will be interpreted :
add(null, null)
 (Otm) break null
break is accepted !
 (Otm) list
break Nr. : 1
null
 (Otm) run
break Nr. 1 is reached !
the next term, which will be interpreteted :
null
 (Otm) where
add(null, null)
 (Otm) list
break Nr. : 1
null
 (Otm) delete 1
break is deleted !
 (Otm) step
you are again in the CASE-term
 with evaluation Nr. : 1
CASE i2
OF
null:
i1;
succ(i2'):
succ(add(i1, i2'));
ESAC
this CASE-term is enclosed by the term :
add(null, null)
 (Otm) step
the next term, which will be interpreted :
i1[null]
 (Otm) step
the next term, which will be interpreteted :
null
 (Otm) step
```

```
the evaluation of :
add(null, null)
has been finished and delivers ->
null
do acknowledge !
(press enter key, if you want to go on)
The term reduction has finished and delivers the result :
null
You are leaving the OBSCURE trace mechanism !

;-) quit
```

5.2 Example queries to the module database

The following text explains some examples of queries to the module database of the user 'arbor'.

He/She wants to know for which specification modules there is an entry into his/her personal database 'arbor'. This query looks as follows:

```
*************************** Select from MDB *****************************
  C-i  start the search in the data base
  C-c  start selecting field names       ESC  escape from selecting field
                                              names
  RET  select a field name               C-t  toggle all fields
  SPC  move to the next field name       DEL  move to the previous field
                                              name
*************************************************************************
SELECT FROM
 Data base: arbor
 List of field names: mname
      iparams [ ] eparams  [ ] isorts  [ ] esorts   [ ]
      iopns   [ ] eopns    [ ] iconstr [ ] econstr  [ ]
      iaxioms [ ] eaxioms  [ ] uses    [ ] isusedby [ ]  compiled [ ]
WHERE Condition: mname LIKE "*"
```

The answer to this query might, for example, look as follows:

```
## Start
arbor.mname:
pair

=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=

arbor.mname:
list

=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=

arbor.mname:
set

=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=

arbor.mname:
set_of_pair

## The End
```

Hence, the personal module database of the user **arbor** contains four specifications with the names *pair*, *list*, *set* and *set_of_pair* respectively.

As a further example, the user wants to display the name of all specification modules, which have been changed since their entry into the module database, but have not been recompiled. The query looks as follows:

```
*************************** Select from MDB *****************************
  C-i  start the search in the data base
  C-c  start selecting field names       ESC  escape from selecting field
                                              names
  RET  select a field name               C-t  toggle all fields
  SPC  move to the next field name       DEL  move to the previous field
                                              name
*************************************************************************

SELECT FROM
 Data base: arbor
 List of field names: mname
      iparams [ ]  eparams  [ ]  isorts  [ ]  esorts   [ ]
      iopns   [ ]  eopns    [ ]  iconstr [ ]  econstr  [ ]
      iaxioms [ ]  eaxioms  [ ]  uses    [ ]  isusedby [ ]  compiled [ ]
WHERE Condition: compiled LIKE "false"
```

The following query asks for the name and the list of imported sorts of all specifications importing the sort '`stack`'.

```
*************************** Select from MDB *****************************
  C-i  start the search in the data base
  C-c  start selecting field names       ESC  escape from selecting field
                                              names
  RET  select a field name               C-t  toggle all fields
  SPC  move to the next field name       DEL  move to the previous field
                                              name
*************************************************************************

SELECT FROM
 Data base: arbor
 List of field names: mname
      iparams [ ]  eparams  [ ]  isorts  [X]  esorts   [ ]
      iopns   [ ]  eopns    [ ]  iconstr [ ]  econstr  [ ]
      iaxioms [ ]  eaxioms  [ ]  uses    [ ]  isusedby [ ]  compiled [ ]
WHERE Condition: stack IN isorts
```

The last example asks for the name and the list of the exported sorts of all specifications fulfilling the following condition: the list of imported sorts is equal to the list of exported sorts.

```
*************************** Select from MDB ******************************
  C-i  start the search in the data base
  C-c  start selecting field names       ESC  escape from selecting field
                                              names
  RET  select a field name               C-t  toggle all fields
  SPC  move to the next field name       DEL  move to the previous field
                                              name
*************************************************************************

SELECT FROM
 Data base: arbor
 List of field names: mname
      iparams [ ]  eparams  [ ]  isorts  [ ]  esorts   [X]
      iopns   [ ]  eopns    [ ]  iconstr [ ]  econstr  [ ]
      iaxioms [ ]  eaxioms  [ ]  uses    [ ]  isusedby [ ]  compiled [ ]
WHERE Condition: isorts=esorts
```

Chapter 6

OBSCURE and UNIX

The following sections describe the call of the components of the OBSCURE system from a standard UNIX shell. All component programs can be found in the subdirectory '`d-run`' of the OBSCURE system.

6.1 Calling the parser

The parser is implemented by three programs in the OBSCURE system:

- a syntactical analyzer, called '`parser`';
- a program, called '`mdb-parser`' inserting the *.O* file created by the parser into a module database;
- a program, called '`compile-command`' which combines the two parts preceding into one program.

  The program '`compile-command`' is called by the '`o-parser`' command of the '`obscure-mode`

The description of these three programs is given as follows:

(i) **Name** *parser*

Action parsing of a specification and creation of a '`.O`' file.

Synopsis parser [-efv] '**`filename`**'

Description The file '**`filename`**' is parsed and an internal representation is created. The name of the file must end in '`.T`'. In the file created the extension is changed into '`.O`'. The filename given to the parser as a parameter may either have the extension '`.T`' or not. In the latter case the extension is supplied by the parser.

Options

-e The text on which the parser is working is displayed during the parsing.

-f The standard output is redirected to '`FILENAME.E`' (instead of the screen).

-v The version number of the parser is displayed.

FILES ~obscure/d-run/parser

(ii) **Name** *mdb-parser*

Action enters the internal representation of a specification created by the parser into the personal module database.

Synopsis mdb-parser '**`filename`**'

Description The file '**`filename`**' is entered into the module database. The name of the file must have the extension '`.O`' and the original file is deleted at the end of the process, so that the file does not exist in two incarnations. As above, the name can be given either with or without the extension.

FILES ~obscure/d-run/mdb-parser

(iii) **Name** *compile-command*

Action parsing of a specification and writing its internal representation into the personal module database.

Synopsis compile-command [-efv] '**`filename`**'

Description The file '**`filename`**' is parsed and entered into the personal module database. The name of the file must end in '`.T`'. As above the filename given to the parser as a parameter may have the extension '`.T`' or not.

Options

-e The text on which the parser is working is displayed during the parsing.

-f The standard output is redirected to '`FILENAME.E`' (instead of the screen).

-v The version number of the parser is displayed.

FILES
~obscure/d-run/parser

~obscure/d-run/mdb-parser

~obscure/d-run/compile-command

6.2 Calling the interpreter

Name *Interpreter*

Action The *interpreter mode* for the interpretation of specifications is started.

Synopsis interpreter [‘`filename`’]

Description The interpreter environment is started and the specification given by ‘`filename`’ is loaded. For a complete description of the interpreter environment see 4.4.1 [The Interpreter Mode], page 66.

FILES ~obscure/d-run/interpreter

6.3 Calling the Source-to-Source-Translation

Name *sotosotra*

Action The source to source translation of specifications into C is started.

Synopsis sotosotra ‘`filename`’

Description The specification ‘`filename`’ is read from the personal database and translated into a C program with the same name. A header file ‘`.h`’ and a C file ‘`.c`’ are created.

FILES ~obscure/d-run/sotosotra

6.4 Calling the module database

There are three programs allowing to work with the module databases.

(i) **Name** *mdb-install*

Action Installing a personal module database.

Synopsis mdb-install

Description A personal module database is installed. The module database has the name contained by the environment variable *USER*.

FILES ~obscure/d-run/mdb-install

(ii) **Name** *mdb-select*

Action A query to a module database is executed.

Synopsis mdb-select [-m'`filename`' -f'`list of field names`' -t'`database name`' -c'`search condition`']

Description This program starts queries to a module database. The queries are directed by the options.

Options

-m *output file*

The answer of the query is written into this file. The default value is '`stdin`'.

-f *list of field names*

The contents of the field names of the list will be outputted. For more information on the organization of the module database see 4.3.2 [The OBSCURE module database], page 58.

-t *database name*

The name of the database to search in. The default value is the name given in the environment variable *USER*.

-c *conditions*

The condition of the search has to be given here. Please remember to escape symbols recognized by the shell, like * or " by a backslash, i.e. write * instead of *. For more information on the syntax of the conditions allowed see 4.3.5 [The third query parameter `<condition>`], page 63.

FILES ~obscure/d-run/mdb-select

(iii) **Name** *mdb-delete*

Action Deleting a specification from a module database.

Synopsis mdb-delete [-m'`filename`' -s'`specificationname`' -t'`databasename`' -v]

Description A specification is deleted from the database.

Options

-m *output file*

The names of the specifications deleted are written to this file. The default value is '`stdout`'.

-s *specification name*
The name of the specification to be deleted must be given here. There is no default name and an error message will be printed if no name is given.

-t *database name* The database, from which the file shall be deleted, has to be given here. The default value is the name given in the environment variable *USER*.

-v This option writes the names of the specifications deleted into the file identified by the `-m` option.

FILES ~obscure/d-run/mdb-delete

Appendix A

Index of commands and functions

For each command the page where it is explained is indicated.

R

S

Appendix B

Index of variables

O

Appendix C

The context free syntax for the specification language

The context free syntax for the specification language is given in *Extended Backus Naur Form* (EBNF). Nonterminal symbols are enclosed by ‘<’ and ‘>’. Repetitions are indicated by { and }; for instance { `<list>` } stands for zero or more occurrences of the nonterminal <list>. Optional parts are enclosed by [and]. The keywords of the language, like **SORTS** are printed in bold face.

The grammar rules are:

(i) Specifications

```
<composed specification>    ::= <simple specification>
                              | <composed specification>
                                PLUS <simple specification>
                              | <composed specification>
                                COMPOSE <simple specification>
                              | <composed specification>
                                X_COMPOSE <simple specification>
                              | <composed specification>
                                F_COMPOSE <simple specification>

<simple specification>      ::= INCLUDE <name>
                              | <atomic specification>
                              | <simple specification> FORGET
                                <list of sorts and operations>
                              | <simple specification>
                                FORGET_ALL_BUT
                                <list of sorts and operations>
                              | <simple specification> E_RENAME
                                <rename list>
                              | <simple specification> I_RENAME
```

```
                                 <rename list>
                               | <simple specification> E_AXIOMS
                                 <axiom> ENDAXIOMS
                               | <simple specification> I_AXIOMS
                                 <axiom> ENDAXIOMS
                               | <simple specification> SUBSET OF
                                 <sort> BY <axiom> ENDSUBSET
                               | <simple specification> QUOTIENT OF
                                 <sort> BY <axiom> ENDQUOTIENT
                               | (<composed specification>)
```

(ii) Atomic Specifications

```
<atomic specification>       ::= IMPORTS
                                 <list of sorts and operations>
                                 [ CREATE
                                   <list of sorts and operations>
                                   SEMANTICS
                                   <algorithmic semantics>
                                   ENDCREATE ]
                               | CREATE
                                 <list of sorts and operations>
                                 SEMANTICS <algorithmic semantics>
                                 ENDCREATE

<algorithmic semantics>      ::= CONSTRS <list of constructors>
                                 [ [ WITH IMPORTED CONSTRS
                                     <list of constructors> ]
                                   VARS <list of variables>
                                   PROGRAMS <list of programs> ]
                               | [ WITH IMPORTED CONSTRS
                                   <list of constructors> ]
                                 VARS <list of variables>
                                 PROGRAMS <list of programs>

<list of constructors>       ::= { <constructor>[,] }

<constructor>                ::= <name>:-> <sort>
                               | <name>: <args> {[,] <args> } -> <sort>
                               | <name> _: <args> -> <sort>
                               | _ <name> _: <args>[,] <args> -> <sort>
                               | <name> _ <name> { , <name> }:
                                 <args> {[,] <args> } -> <sort>
                               | _ <name> { _ <name> }:
                                 <args> {[,] <args> } -> <sort>

<args>                       ::= [ LAZY ] <sort>
```

```
<list of variables>        ::= {{ <name>[,] }: <sort>[,] }

<list of programs>         ::= <head> <- <term>;
                               { <head> <- <term>; }

<head>                     ::= <prefix name>
                             | <prefix name> ({ <variable>, })
                             | <infix name> <variable>
                             | <variable> <infix name> <variable>
                             | <mixfix name> <variable> <mixfix name>
                               { <variable> <mixfix name> }
                             | <variable> <mixfix name>
                               { <variable> <mixfix name> }

<prefix name>              ::= <name>

<infix name>               ::= <name>

<mixfix name>              ::= <name>

<variable>                 ::= <name>

<term>                     ::= <infix term>
                             | <mixfix name> <infix term>
                               <mixfix name>
                               { <infix term> <mixfix name> }

<infix term>               ::= <baseterm>
                             | <infix term> <infix name> <baseterm>

<baseterm>                 ::= (<term>)
                             | ERROR (<sort>)
                             | <variable>
                             | IF <term> THEN <term> ELSE
                               <term> FI
                             | CASE <term> OF
                               <head>: <term>; { <head>: <term>; }
                               [ ELSE <term> ] ESAC
                             | <prefix name>
                             | <prefix name> (<term> { , <term> })
                             | <infix name> <baseterm>

<name>                     ::= <letter> { <symbol> }
                             | <special symbol>

<letter>                   ::= a | b | c | ... | z | A | B | ... | Z
```

```
<symbol>                          ::= <letter> | <digit> | _
                                    | <special symbol>

<digit>                           ::= 0 | 1 | 2 | ... | 9

<special symbol>                  ::= < | > | = | # | ^ | [ | ] |   |   | +
                                    | * | / | - | \  | @ | $ | %
```

(iii) Sorts and operations

```
<list of sorts and operations> ::= SORTS <list of sorts>
                                   [ OPNS <list of operations> ]
                                 | OPNS <list of operations>

<list of sorts>                ::= { <sort>[,] }

<sort>                         ::= <name>

<list of operations>           ::= { <operation>[,] }

<operation>                    ::= <name>:-> <sort>
                                 | <name>: <sort> {[,] <sort>} -> <sort>
                                 | <name> _: <sort> -> <sort>
                                 | _ <name> _: <sort>[,] <sort>
                                   -> <sort>
                                 | <name> _ <name> { _ <name> }:
                                      <sort> {[,] <sort> } -> <sort>
                                 | _ <name> { _ <name> }:
                                      <sort> {[,] <sort> } -> <sort>
```

(iv) Renamings

```
<rename list>                  ::= SORTS { <sort>, } AS SORTS { <sort>,}
                                   [ OPNS { <operation>, } AS OPNS
                                     { <operation name>, } ]
                                 | AS SORTS { <sort>, } OPNS
                                   { <operation name>, }

<operation name>               ::= <name>
                                 | <name> _
                                 | _ <name> _
                                 | <name> _ <name> { _ <name> }
                                 | _<name> { _ <name> }
```

(v) Formulas

```
<axiom>              ::= <formula>; { <formula>; }
                       | VARS <list of variables>;
                         <formula>; { <formula>; }

<formula>            ::= <disjunction>
                       | <disjunction> { => <formula> }
                       | <disjunction> { <=> <formula> }

<disjunction>        ::= <conjunction> { — <conjunction> }

<conjunction>        ::= <simple formula> { & <simple formula>}

<simple formula>     ::= <equation>
                       | (<formula>)
                       | ! <simple formula>
                       | EX <variable>, { <variable>, }.
                         <simple formula>
                       | ALL <variable>, { <variable>, }.
                         <simple formula>

<equation>           ::= <term> == <term>
                       | <term> [= <term>
```

Remarks

1. In order to save brackets when writing terms, equations and formulas the following priorities are valid:

- the operators '=>' and '<=>' have the lowest priority and are left associative;
- the disjunction '|' has the next higher priority;
- the conjunction '&' has a higher priority than the disjunction;
- on the next higher level of priority are '!', 'EX' and 'ALL';
- the operators '==' and '[=' have the highest priority.

2. The following sorts and operation are called *basic*:

sort	bool
operations	true, false, =, and, or, not
sort	integer
operations	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, =, <, <=, >=, >, +, *, -, div, mod

Basic sorts and operations have not to appear on the list of imported sorts and operations of a specification. Moreover, their meaning is "known" to the interpreter.

Note that you have to express numbers higher than 10 as arithmetic formulas in the specifications and in the interpreter. The interpreter generates results greater than 10 as usual.

3. An exhaustive description of the context conditions of the specification language can be found in [Zey 89]

Appendix D

A more complex example

As a more complex example, the notion of a signature is expressed by a specification. While the example itself is an academic rather than a real-life example, the goal of this appendix is to illustrate the design of a non-trivial specification in the specification language. The design of the specification has been done with the help of the OBSCURE system.

First the overall specification of a signature is given so that the reader gets an idea of what is specified in this example. Intuitively a signature consists of a set of sort names and a set of operations. Actually, a slightly different view is taken here. The set of operations is divided into two sets, a set of constructors and a set of "defined" operations. An operation consists of an operation name, a (possibly empty) list of source sorts and a target sort. The specifications of operations and of symbols, which are used for operations and sort names are left to the reader. According to what has been said above, a signature is a pair of collections of operations together with some operations on signatures like adding a constructor to a specification, checking whether an operation is a "defined" operation of the signature, etc. This is expressed in the specification '`construct_signature.T`'.

Some of the data types imported by the specifications are not detailed here, and it is assumed that the reader is able to specify them.

```
## The specification  construct_signature

IMPORTS
 SORTS
  pair_of_collection_of_operation
  collection_of_operation
  operation        ## the specification of this sort is left to the reader.
  symbol
  list_of_symbol
```

OPNS

```
## operations of pair_of_collection_of_operation
pair_of_collection_of_operation_with
_ as_constructors
_ as_defined_operators :
            collection_of_operation  collection_of_operation ->
                 pair_of_collection_of_operation
constructors_of _      : pair_of_collection_of_operation ->
                                 collection_of_operation
defined_operators_of _ : pair_of_collection_of_operation ->
                                 collection_of_operation

## operations of collection_of_operation
empty_collection_of_operation :
               -> collection_of_operation
_ is_empty     : collection_of_operation -> bool
_ is_in _      :               operation
                 collection_of_operation -> bool
one_of _       : collection_of_operation -> operation
_ with    _    : collection_of_operation operation
                    -> collection_of_operation
_ without _    : collection_of_operation operation
                    -> collection_of_operation
## conjunction operator
_ u _          : collection_of_operation collection_of_operation
                    -> collection_of_operation
## subset operator
_ c_  _        : collection_of_operation collection_of_operation
                    -> bool
## target sort of the operation with name symbol and target sorts
## list of symbols
sort__of _ with_source _ in _ as_opns :
  symbol list_of_symbol collection_of_operation -> symbol
op_symbol _ with_source _ occures_in _ as_opns :
  symbol list_of_symbol collection_of_operation  -> bool
## are there any operations excluding each other ?
_ is_compatible_to _ :
  collection_of_operation collection_of_operation -> bool
## are there operations with the same name and same source sorts ?
_ has_a_common_op_symbol_with_same_source_with _ :
                             collection_of_operation
                             collection_of_operation -> bool

## operations of operation
name_of _      : operation -> symbol
source_of _    : operation -> list_of_symbol
target_of _    : operation -> symbol
```

```
## operations of symbol
_  =  _       : symbol symbol -> bool

## operations of list_of_symbol
last_of _     : list_of_symbol -> symbol
body_of _     : list_of_symbol -> list_of_symbol
_ = _         : list_of_symbol list_of_symbol -> bool
```

CREATE

OPNS

```
## Intended to become the empty signature
empty_pair_of_collection_of_operation
                          : -> pair_of_collection_of_operation
## Adds a constructor to the constructor set while checking whether
## this is possible.
_ with_constructor   _    : pair_of_collection_of_operation
              operation -> pair_of_collection_of_operation

## Adds a defined operator to the set of defined operators while
## checking whether this is possible.
_ with_defined_operator _ : pair_of_collection_of_operation
              operation -> pair_of_collection_of_operation

## Removes a constructor.
_ without_constructor _        :    pair_of_collection_of_operation
                     operation -> pair_of_collection_of_operation
## Removes a defined operator.
_ without_defined_operator _   :    pair_of_collection_of_operation
                     operation -> pair_of_collection_of_operation
## Gets a constructor from the constructor set.
one_constructor_of _           :    pair_of_collection_of_operation
                               -> operation
## Gets a defined operator from the set of defined operators.
one_defined_operator_of _      :    pair_of_collection_of_operation
                               -> operation
## Checks whether the operation is a defined operator of the signature
_ is_defined_operator_of _     :    operation
                                    pair_of_collection_of_operation
                               -> bool
## Computes the constructor signature, i.e. the signature without the
## set of defined operators
constructor_signature_of _     :    pair_of_collection_of_operation
                               -> pair_of_collection_of_operation
## Checks whether two signatures are "compatible" with each other. For
a better
## understanding of the notion "compatible", please read the semantics
## of this operation.
```

```
_ is_compatible_to _       : pair_of_collection_of_operation
                             pair_of_collection_of_operation -> bool
## Joins two signatures if possible.
_ u _                      : pair_of_collection_of_operation
                             pair_of_collection_of_operation
                          -> pair_of_collection_of_operation
## Checks whether one signature is the subset of another signature.
_ c_ _                     : pair_of_collection_of_operation
                             pair_of_collection_of_operation -> bool
## Finds the target sort of the operation with the name "symbol" and source
## sorts "list_of_symbol".
sort__of _ with_source _ in _ as_sigma :
                             symbol list_of_symbol
                             pair_of_collection_of_operation -> symbol
## Checks whether the operation with the name "symbol" and source sorts
## "list_of_symbol" occurs in the signature.
op_symbol _ with_source _ occurs_in _ as_sigma :
                             symbol list_of_symbol
                             pair_of_collection_of_operation -> bool
## Checks whether the operation with the name "symbol" and source sorts
## "list_of_symbol" is a constructor of the signature.
op_symbol _ with_source _ is_constructor_in _ as_sigma :
                             symbol list_of_symbol
                             pair_of_collection_of_operation -> bool
## Checks whether the operation with the name "symbol" and source sorts
## "list_of_symbol" is a defined operator of the signature.
op_symbol _ with_source _ is_defined_operator_in _ as_sigma :
                             symbol list_of_symbol
                             pair_of_collection_of_operation -> bool
```

SEMANTICS

WITH IMPORTED CONSTRS

```
pair_of_collection_of_operation_with
_ as_constructors
_ as_defined_operators :
             collection_of_operation  collection_of_operation ->
                                 pair_of_collection_of_operation
```

VARS

```
sig1 sig2      : pair_of_collection_of_operation
opcol1 opcol2  : collection_of_operation
opnam1         : symbol
sortlist1      : list_of_symbol
op1 op2        : operation
```

PROGRAMS

```
empty_pair_of_collection_of_operation <-
   pair_of_collection_of_operation_with
          empty_collection_of_operation as_constructors
          empty_collection_of_operation as_defined_operators ;

sig1 with_constructor op1 <-
  ## A constructor may not be a defined operator, too.
  IF     op_symbol ( name_of op1 )
         with_source ( source_of op1 )
         occures_in ( defined_operators_of sig1 ) as_opns
  THEN   ERROR(pair_of_collection_of_operation)
  ELSE   pair_of_collection_of_operation_with
          ( ( constructors_of sig1 )
            with
            op1 )                        as_constructors
          ( defined_operators_of sig1 )  as_defined_operators
  FI;

sig1 with_defined_operator op1 <-
  ## A defined operator may not be a constructor, too.
  IF     op_symbol ( name_of op1 )
         with_source ( source_of op1 )
         occures_in ( constructors_of sig1 ) as_opns
  THEN   ERROR(pair_of_collection_of_operation)
  ELSE   pair_of_collection_of_operation_with
          ( constructors_of sig1 ) as_constructors
          ( ( defined_operators_of sig1 )
            with
            op1
          )                        as_defined_operators
  FI;

sig1 without_constructor op1  <-
   pair_of_collection_of_operation_with
   ( ( constructors_of sig1      ) without op1 ) as_constructors
   (   defined_operators_of sig1               ) as_defined_operators;

sig1 without_defined_operator op1  <-
   pair_of_collection_of_operation_with
   (   constructors_of sig1                    ) as_constructors
   ( ( defined_operators_of sig1 ) without op1 ) as_defined_operators;

one_constructor_of sig1  <-
   one_of ( constructors_of sig1 ) ;
```

```
one_defined_operator_of sig1 <-
   one_of ( defined_operators_of sig1 ) ;

op1 is_defined_operator_of sig1 <-
   op1 is_in (defined_operators_of sig1);

constructor_signature_of sig1  <-
   pair_of_collection_of_operation_with
     ( constructors_of sig1       ) as_constructors
     empty_collection_of_operation  as_defined_operators ;

## Two operations with the same name and the same source sorts must not
## exist in the constructors of sig1 and the defined operators of sig2
## and vice versa. The operations of the two constructor signatures and
## the two signatures of the defined operators must be compatible with
## each other. The semantics of this compatibility will be explained
## later.
sig1 is_compatible_to sig2  <-
   ( not
     ( (constructors_of sig1)
       has_a_common_op_symbol_with_same_source_with
       (defined_operators_of sig2)
     )
     or
     ( (constructors_of sig2)
       has_a_common_op_symbol_with_same_source_with
       (defined_operators_of sig1)
     )
   )
   and
   ( ( constructors_of sig1 )
     is_compatible_to
     ( constructors_of sig2 )
   )
   and
   ( ( defined_operators_of sig1 )
     is_compatible_to
     ( defined_operators_of sig2 )
   ) ;

sig1 u sig2   <-
  IF   sig1 is_compatible_to sig2
  THEN pair_of_collection_of_operation_with
         ( ( constructors_of sig1 ) u ( constructors_of sig2 ) )
         as_constructors
         ( ( defined_operators_of sig1 )
           u ( defined_operators_of sig2 ) )
         as_defined_operators
```

```
    ELSE ERROR(pair_of_collection_of_operation)
    FI;

  sig1 c_ sig2  <-
    ( (constructors_of sig1)       c_ (constructors_of sig2) )        and
    ( (defined_operators_of sig1 ) c_ (defined_operators_of sig2) );

  sort__of opnam1 with_source sortlist1 in sig1 as_sigma <-
    IF   op_symbol opnam1
         with_source sortlist1
         is_constructor_in sig1 as_sigma
    THEN sort__of    opnam1
         with_source sortlist1
         in          (constructors_of (sig1)) as_opns
    ELSE
      IF   op_symbol   opnam1
           with_source sortlist1
           is_defined_operator_in sig1 as_sigma
      THEN sort__of    opnam1
           with_source sortlist1
           in          (defined_operators_of (sig1)) as_opns
      ELSE ERROR(symbol)
      FI
    FI;

  op_symbol opnam1 with_source sortlist1 occurs_in sig1
            as_sigma <-
    (op_symbol opnam1 with_source sortlist1
     is_constructor_in sig1 as_sigma)
    or
    (op_symbol opnam1 with_source sortlist1
     is_defined_operator_in sig1 as_sigma);

  op_symbol opnam1 with_source sortlist1
     is_constructor_in sig1 as_sigma <-
    op_symbol   opnam1
    with_source sortlist1
    occures_in  (constructors_of sig1) as_opns ;

 op_symbol opnam1 with_source sortlist1
    is_defined_operator_in sig1 as_sigma <-
    op_symbol   opnam1
    with_source sortlist1
    occures_in  (defined_operators_of sig1) as_opns


ENDCREATE
```

The sort `pair_of_collection_of_operation` which is intended to become the sort `signature` will be defined in the next specifications by a parameterization of the sort `pair` with the sort `collection_of_operation`.

```
## The specification   pair_of_collection_of_operation

INCLUDE pair
  I_RENAME
        SORTS   firstsort              secondsort
     AS SORTS   collection_of_operation collection_of_operation

  E_RENAME
    SORTS   pair
    OPNS    pair _ \  _ endpair : collection_of_operation
                                  collection_of_operation    -> pair
            first _   : pair -> collection_of_operation
            second _  : pair -> collection_of_operation
    AS
    SORTS   pair_of_collection_of_operation
    OPNS    pair_of_collection_of_operation_with
            _ as_constructors _ as_defined_operators,
            constructors_of _ ,
            defined_operators_of _
```

The specification of the sort `pair` is trivial and left to the reader. The specification of the sort `collection_of_operation` is more interesting and will be explained in more detail. Essentially, it is a parameterization of the sort `set_of_operation`, which is left to the reader (the specification of set should be simple by now; operations are triples of `symbol`, `list_of_symbol` and `symbol`, standing for the name, the source sorts and the target sort respectively). But a set of operations does not capture the fact that operations in a collection of operations, as it is needed for our notion of signature, should be "compatible" with each other. For example there should not be two operations with the same name, the same arity, the same source sorts, but different target sorts. The specification of the sort `collection_of_operation` is done in three steps. First the sort `set` is parameterized by the sort `operation`, then this specification is composed with a specification defining new operators to build collections, which check the compatibility, and finally, the old constructors are forgotten so that only collections containing compatible operations can be created.

```
## The specification set_of_operation


INCLUDE set
  I_RENAME
       SORTS el
    AS SORTS operation

  E_RENAME
       SORTS set
       OPNS  empty_set : -> set
    AS SORTS set_of_operation
```

```
OPNS  empty_set_of_operation
```

The following specification introduces those operations which construct correct collections of operations.

```
## The specification  construct_collection_of_operation

IMPORTS
  SORTS
    set_of_operation
    operation
    symbol
    list_of_symbol

 OPNS
  ## operations of set_of_operation
  empty_set_of_operation : -> set_of_operation
  _ is_empty       : set_of_operation -> bool
  one_of _         : set_of_operation -> operation
  _ with    _      : set_of_operation operation -> set_of_operation
  _ c_ _           : set_of_operation set_of_operation -> bool
  _ u  _           : set_of_operation set_of_operation -> set_of_operation
  _ without _      : set_of_operation operation -> set_of_operation
  ## operations of operation
  name_of _        : operation -> symbol
  source_of _      : operation -> list_of_symbol
  target_of _      : operation -> symbol
  ## operations of symbol
  _   =   _        : symbol symbol -> bool
  ## operations of list_of_symbol
  last_of _        : list_of_symbol -> symbol
  body_of _        : list_of_symbol -> list_of_symbol
  _ = _            : list_of_symbol list_of_symbol -> bool

CREATE
  OPNS
  ## the following operations are intended as a replacement for the usual
  ## operations on sets.
  _ with_op  _    : set_of_operation operation -> set_of_operation
  _ u_op     _    : set_of_operation set_of_operation
                                                 -> set_of_operation
  sort__of _ with_source _ in _ as_opns :
    symbol list_of_symbol set_of_operation -> symbol
  op_symbol _ with_source _ occures_in _ as_opns :
    symbol list_of_symbol set_of_operation  -> bool
  _ is_compatible_to _ :
    set_of_operation set_of_operation -> bool
  _ has_a_common_op_symbol_with_same_source_with _ :
```

```
                              set_of_operation
                              set_of_operation -> bool
```

SEMANTICS

VARS

```
  opset1 opset2   : set_of_operation
  op1 op2         : operation
  opsymb1 opsymb2 : symbol
  source1 source2 : list_of_symbol
  opcol1  opcol2  : set_of_operation
```

PROGRAMS

```
 opcol1 with_op op1 <-
   IF   op_symbol (name_of op1) with_source (source_of op1)
        occures_in opcol1 as_opns
   THEN
     IF   ( sort__of (name_of op1)
            with_source (source_of op1)
            in opcol1 as_opns )
          =
          ( target_of op1 )
     THEN opcol1
     ELSE ERROR(set_of_operation)
     FI
   ELSE opcol1 with op1
   FI;

 opcol1 u_op opcol2  <-
   IF   opcol1 is_compatible_to opcol1
   THEN opcol1 u opcol2
   ELSE ERROR(set_of_operation)
   FI;

 sort__of opsymb1 with_source source1 in opcol1 as_opns <-
   IF   opcol1 is_empty
   THEN ERROR(symbol)
   ELSE
     IF   ( name_of   (one_of opcol1) = opsymb1 )  and
          ( source_of (one_of opcol1) = source1 )
     THEN   target_of (one_of opcol1)
     ELSE sort__of opsymb1 with_source source1 in
          ( opcol1 without (one_of opcol1) )   as_opns
     FI
   FI;

 op_symbol opsymb1 with_source source1 occures_in opcol1 as_opns <-
```

```
  IF   opcol1 is_empty
  THEN false
  ELSE
    IF   ( name_of   (one_of opcol1) = opsymb1 ) and
         ( source_of (one_of opcol1) = source1 )
    THEN true
    ELSE op_symbol opsymb1 with_source source1 occures_in
         ( opcol1 without (one_of opcol1) ) as_opns
    FI
  FI;

opcol1 is_compatible_to opcol2  <-
  IF   opcol1 is_empty
  THEN true
  ELSE
    IF   op_symbol
         ( name_of (one_of opcol1) )
         with_source
         ( source_of (one_of opcol1) )
         occures_in
         opcol2
         as_opns
    THEN
      IF   ( target_of (one_of opcol1) )
           =
           ( sort__of ( name_of ( one_of opcol1 ) )
             with_source ( source_of ( one_of opcol1 ) )
             in opcol2 as_opns
           )
      THEN ( opcol1 without (one_of opcol1) ) is_compatible_to opcol2
      ELSE false
      FI
    ELSE ( opcol1 without (one_of opcol1) ) is_compatible_to opcol2
    FI
  FI;

opcol1 has_a_common_op_symbol_with_same_source_with opcol2 <-
  IF   opcol1 is_empty
  THEN true
  ELSE
    IF   op_symbol   (name_of   (one_of opcol1))
         with_source (source_of (one_of opcol1))
         occures_in  opcol2
         as_opns
    THEN true
    ELSE (opcol1 without (one_of opcol1))
         has_a_common_op_symbol_with_same_source_with
         opcol2
```

```
    FI
  FI
ENDCREATE
```

In the third and last specification the specification construct_collection_of_operation is composed with the specification set_of_operation. The sort set_of_operation is renamed to collection_of_operation and the old constructors are forgotten. The main property of this new sort is then expressed by an export axiom.

```
## The specification  collection_of_operation

(
INCLUDE construct_collection_of_operation

X_COMPOSE

INCLUDE set_of_operation
)

FORGET
 OPNS
  _ with    _    : set_of_operation operation -> set_of_operation
  _ u  _         : set_of_operation set_of_operation -> set_of_operation

E_RENAME
    SORTS  set_of_operation
    OPNS   empty_set_of_operation
                        : -> set_of_operation
           _ with_op  _ : set_of_operation operation
                             -> set_of_operation
           _ u_op     _ : set_of_operation set_of_operation
                                         -> set_of_operation
 AS SORTS  collection_of_operation
    OPNS   empty_collection_of_operation ,
           _ with _ ,
           _ u _


E_AXIOMS

 VARS col : collection_of_operation,
      op  : operation;

   ( op is_in col                                   ) == true

   =>

   ( op_symbol  (name_of op) with_source (source_of op)
```

```
    occures_in (col without op) as_opns                ) == false
```

ENDAXIOMS

In the last specification **signature**, the parts specified up to now are composed and the sort **collection_of_operation** is forgotten, because it was needed only for technical purposes. Then the sort **pair_of_collection_of_operation** is renamed to **signature**. The main property of this new sort is expressed by an export axiom.

```
## The specification  signature

(
INCLUDE construct_signature

X_COMPOSE

INCLUDE pair_of_collection_of_operation

X_COMPOSE

INCLUDE collection_of_operation
)

FORGET
 SORTS
  collection_of_operation

E_RENAME
      SORTS pair_of_collection_of_operation
      OPNS  empty_pair_of_collection_of_operation : ->
                pair_of_collection_of_operation
   AS SORTS signature
      OPNS  empty_signature

E_AXIOMS
    VARS sig1 : signature,
         nam1 : symbol,
         lst1 : list_of_symbol;
      ! ( ( op_symbol nam1 with_source lst1
            is_constructor_in sig1 as_sigma )
          == true
          &
          ( op_symbol nam1 with_source lst1
            is_defined_operator_in sig1 as_sigma )
          == true
        )
ENDAXIOMS
```

Appendix E

Installation guide

E.1 Installation of the system

E.1.1 If shipped together with Emacs

The user interface of the OBSCURE system is programmed in *GNU-Emacs Lisp*. A part of *GNU-Emacs* Version 18.54 is also on the tape. A complete *Emacs* version can be sent on request.

When installing the *Emacs* adaption, the following should be observed:

- *Emacs* should be installed in the directory '`/users/obscure/emacs`'.
- The *Emacs* implementation of the OBSCURE interface must be saved onto the file: '`/users/obscure/emacs/lisp/obscure.el`'.
- Every user of the OBSCURE system must add the following lines to his/her personal '`~/.emacs`' file:

```
     ; ------------------------------------------------------------------
(setq exec-path (cons "/users/obscure/emacs/"
      (cons "/users/obscure/emacs/etc/" exec-path)))
(setq Info-directory "/users/obscure/emacs/info/")
(setq load-path (cons "/users/obscure/emacs/lisp/" load-path))
(setq auto-mode-alist (cons '("\ \ .T$" . obscure-mode)
      auto-mode-alist))
(autoload 'obscure-mode "obscure" " " t)
(setq obscure-mode-hook
  '(lambda () (setq o-vip-desired nil)))
; ------------------------------------------------------------------
```

E.1.2 If not shipped together with Emacs

The user interface of the OBSCURE system is programmed in *GNU-Emacs Lisp*. The use of the *GNU-Emacs* Version 18.50 or a later version is neccessary.

When installing the *Emacs* adaption, the following should be observed:

- $EMACS stands for the directory *Emacs* is installed in.
- The *Emacs* implementation of the **OBSCURE** interface can be found in the file '`obscure.el`'.
- It is neccessary to copy the file '`obscure.el`' into the directory '`$EMACS/lisp`' to make it accessible to all users.
- For technical reasons, two small changes had to be made in the file '`$EMACS/lisp/compile.el`'. Please make these changes in your file, too (they do not alter the normal behaviour of the `compile-command`). They are printed at the end of this list.
- In order to make the **OBSCURE** manual accessible interactively, a directory named '`d-obscure`' has to be produced within the directory '`$EMACS/info`'. Then the files '`obscure`' and '`obs1`' to '`obsn`' have to be copied from the directory '`/users/obscure/d-doku`' into the directory '`$EMACS/info/d-obscure`'. Finally, the following line has to be added to '`$EMACS/info/dir`':

  ```
  * obscure: (d-obscure/obscure). The manual of the OBSCURE-System.
  ```

- Every user of the system should add the following lines to his/her personal '`~/.emacs`' file (or create a file with this name and this line):

```
    ; ------------------------------------------------------------------
(setq auto-mode-alist (cons '("\ \ .T$" . obscure-mode)
      auto-mode-alist))
(autoload 'obscure-mode "obscure" " " t)
(setq obscure-mode-hook
  '(lambda () (setq o-vip-desired nil)))
; ------------------------------------------------------------------
```

Please make the following changes in the file '`$EMACS/lisp/compile.el`': Add the following behind the line `(provide 'compile)`:

```
(defvar compilation-sentinel-user-action-wanted nil ; new
 "*If t perform (compilation-sentinel-user-action)  ; new
  after compilation is finished")                   ; new
```

Add the two lines marked by '`new`' to the function `compilation-sentinel`:

```
(defun compilation-sentinel (proc msg)
  (cond ((null (buffer-name (process-buffer proc)))
 ;; buffer killed
 (set-process-buffer proc nil))
```

```
((memq (process-status proc) '(signal exit))
 (let* ((obuf (current-buffer))
omax opoint)
   ;; save-excursion isn't the right thing if
   ;;  process-buffer is current-buffer
   (unwind-protect
       (progn
  ;; Write something in *compilation* and hack its mode line,
  (set-buffer (process-buffer proc))
  (setq omax (point-max) opoint (point))
  (goto-char (point-max))
  (insert ?\ n mode-name " " msg)
  (forward-char -1)
  (insert " at "
  (substring (current-time-string) 0 -5))
  (forward-char 1)
  (setq mode-line-process
        (concat ": "
        (symbol-name (process-status proc))))
  ;; If buffer and mode line will show that the process
  ;; is dead, we can delete it now.  Otherwise it
  ;; will stay around until M-x list-processes.
  (delete-process proc)
  (if compilation-sentinel-user-action-wanted ; new
      (compilation-sentinel-user-action))     ; new
  )
      (setq compilation-process nil)

[rest of function definiton]
```

After these changes have been made, this file has to be recompiled. This is done in *Emacs* by the command `byte-compile-file`. The argument has to be '`compile.el`' (possibly with the pathname).

E.2 Further remarks

It is recommended not to work with the `vip-mode` of *Emacs* and the `obscure-mode` simultaneously.

The *Emacs* command `o-latex-format` can not be used.

The program `obscure-module-graph` is a prototype version and should be ignored.

Example specifications for the automatic translation of specifications into C may be found in the directory '`/users/obscure/d-sot/d-bsp`'.

If changes are made in the file '`obscure.el`' the compiled version '`obscure.elc`' has to be reestablished. This is done in *Emacs* by the command `byte-compile-file` with the argument '`obscure.el`'.

Appendix F

References

[LL 88] Lehmann, T., Loeckx, J., "The specification language of OBSCURE", LNCS **332** pp.131-153 (1988)

[Lo 87] Loeckx, J., "Algorithmic Specifications: A Constructive Specification Method for Abstract Data Types", *TOPLAS* **9**, 4 (1987), pp.646 - 685

[LL 90] Lehmann, T., Loeckx, J., "OBSCURE, a specification language for abstract data types", *Int. Rep. A 19/90, Univ. Saarbrücken* (1990)

[Sta 85] Stallman, R., "GNU Emacs Manual, Third Edition, Emacs Version 18"

[Sto 91] Stolz, M. "Eine verzögerte Auswertung für Algorithmische Spezifikationen—die Theorie und eine Implementierung für OBSCURE", *Master's thesis, Univ. Saarbrücken* (1991), to appear.

[Zey 89] Zeyer, J., "Kontextbedingungen fuer OBSCURE", *Int. Rep. WP 89/11, Univ. Saarbrücken* (1989)