The Early Adaptors Guide to Acoi

Menzo Windhouwer

July 26, 2005

Contents

Chapter 1

Introduction

The aim of this document is to serve as an introductionary guide for setting up the Acoi environment and developing a first feature grammar. By doing so this document also gives an overview of what is really implemented and how this can be used. However, as the implementation is still in progress do not expect all language features to be supported on an equal, stable, level. Alongside the evolvement of the implementation this document has to evolve, so any useful remarks are welcome and may be incorporated during the next iteration.

1.1 Assumptions

For the theory behind Acoi and the feature grammar language you will have to read the scientific publications (see http://monetdb.cwi.nl/~acoi/). This document assumes you have this background knowledge.

The hard- and software context of this document is a personal computer running Linux. If you have the same environment you should be able to follow the steps without many changes. However, in the cases changes are needed you should be able to interpret and adapt the commands to your own specific environment.

The DataBase Management System (DBMS) used for storing the meta-data is MonetDB. In theory it is possible to replace this by another DBMS, but as this has never been done (yet) I will assume you use MonetDB.

1.2 Conventions

The following layout conventions are used for specific information.

Several special sections describe more in-depth information or future plans. They are not important to be able to work with the current implementation, so if you are not interested in this information you may skip them. However, if you

experience problems somewhere they may also give you some hints to their cause. Their layout is as follows:

---

future

To give some insights into where the implementation will or should be going to future ideas are also described or hinted at. And if you, the early adaptor, feel the need to sharpen you hacker/coding skills on some of these future parts/rewrites of Acoi feel free to do so.

To illustrate the steps of installation and usage, dumps of command line sessions or the contents of a file will be shown using the following layout (the three dots (`...`) indicate that not all output or contents are shown):

— *command line*

```
1: (acoi 1) cd ~/acoi
2: (acoi 2) ls
3: CVS   Makefile.ag  acconfig.h  conf          java
 scripts
4: DONE  TODO         bootstrap   configure.ag  rules.mk
 src
5: ...
```

—

The following chapters will now describe the installation and use of Acoi. Good luck and be prepared: there be dragons!

Chapter 2

Installation

2.1 What do you need?

Preceding the installation of Acoi you have to install the following packages (the checkout date or version number corresponds with the Acoi 2.0.0 version of **April 21, 2004**):

Mx and autogen (comming with MonetDB 4.3.16) Mx and autogen are two tools used by the CWI database research group for software development. If you'll be using MonetDB as the database backend to Acoi you'll receive them with the following package. In the other case you need to get them separately, in this case contact the author.

MonetDB (version: 4.3.16) Acoi is developed in a database group whose core technology is the MonetDB database management system (see `http://monetdb.cwi.nl/`). The development environment of Acoi reuses some tools developed for MonetDB. As these tools are not available outside the MonetDB distribution you will have to install MonetDB first. How to do this is described in the `HowToStart` file which you find in the root directory of the distribution. If you managed this it will not be too hard to also install the Acoi distribution.

Gnome XML library (version: 2.6.9) The `HTTP` implementation of the Gnome XML library (see `http://xmlsoft.org/`) is used for WWW downloads, the internal representation of the parse tree and resolving whitebox detector predicates.

Gnome XSLT library (version: 1.1.6) The `XSLT` implementation (see `http://xmlsoft.org/XSLT/`) on top of the Gnome XML library is linked into the FDE.

cSOAP library (version: 1.0.2) Acoi uses SOAP for interaction between the various system components. cSOAP (see http://www.sourceforge.net/projects/csoap/) is an implementation of SOAP on top of the Gnome XML library.

ZEEP(sop) (version: 1.0.0) ZEEP is a cSOAP based SOAP interface on top of MonetDB. It's available as part of the xml module from MonetDB's CVS repository (see http://sourceforge.net/projects/monetdb/).

GNU getopt package You only need this package when your platform doesn't support `getopt_long`, which is by default available at the Linux platform. There is a test for this function in the `configure` script of Acoi, if this test fails contact the author to get the package.

Java Developers Kit (version: 1.2 or higher) Java (see http://www.javasoft.com) is needed for the following packages and for the Swing based query interface.

James Clarks XT (version: 20020426a) Although programmed in Java James Clarks `XSLT` processor XT (see http://blnz.com/xt/) is still the fastest around (see http://www.datapower.com/XSLTMark/). Acoi uses it to process queries and answers.

Tcl (version: 8.0 or higher) Parts of the query infrastructure is currently in Tcl (see http://dev.scriptics.com/).

autoconf (version: 2.57), automake (version: 1.6.3) and libtool (version: 1.4.3) MonetDB provides `autogen` which transforms "simple" Makefiles into `automake` Makefiles. When you develop your own feature grammars this tool maybe frequently used. Notice that version mismatches with these tools may provide some nasty problems, which need specific solutions. So when you encounter those contact the author.

In-depth

The following fragment of `~/.bashrc` gives an idea about the settings required for some of the packages:

— *~/.bashrc*

```
 1: ...
 2: export PREFIX="Linux"
 3:
 4: #MonetDB
 5: export MONETDB_PREFIX="$HOME/monet/$PREFIX"
 6: export MONETDBFARM="$HOME/monet.user’"
 8:
 9: export PATH="$MONETDB_PREFIX/bin:$PATH"
10:
11: #Other local packages, e.g. libxml and libxslt
12: export PATH="$HOME/local/$PREFIX/bin:$PATH"
13: export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:
  $HOME/local/$PREFIX/lib"
14: ...
```

—

Acoi is not (yet) public available, but you already have this document so you should also have a tarball containing the Acoi toolset (`acoi.tgz`). Next to this you also need the basic `configure` setup for a developer (`grammars.tgz`). How to install these tarballs is the topic of the next sections.

2.2 Installing Acoi

Assuming that all the needed packages are available in your `PATH`, `PKG_CONFIG_PATH` or `CLASSPATH`. The following commands compile and install Acoi:

— *command line*

```
 1: (acoi 1) cd ~
 2: (acoi 2) gtar xvfz ~/tmp/acoi.tgz
 3: acoi/
 4: ...
 5: acoi/tcl/tools/Makefile.ag
 6: (acoi 3) cd acoi
 7: (acoi 4) bootstrap
 8: /home/acoi/acoi/conf
 9: ...
10: patching file configure
```

```
11: (acoi 5) mkdir Linux
12: (acoi 6) cd Linux
13: (acoi 7) ../configure --prefix=`pwd`
14: creating cache ./config.cache
15: ...
16: creating acoi_config.h
17: (acoi 8) make
18: cd .. && autoheader
19: ...
20: gmake[1]: Leaving directory '/home/acoi/acoi/Linux'
21: (acoi 9) make install
22: Making install in conf
23: ...
24: gmake[1]: Leaving directory `/home/acoi/acoi/Linux'
```

—

If the configure script fails you're missing a compulsory package or you have the wrong version. Hopefully the error message is clear enough.

Well if you downloaded, compiled and installed any Linux software package lately this should look familiar. The extra `bootstrap` command uses the `autogen` tool, developed for MonetDB, to translate the `Makefile.ag` files into `Makefile.am` as used by `automake`.

Add `$HOME/acoi/Linux/bin` to your local `PATH`, so the Acoi binaries can be found.

In-depth

Using this `PATH` the `checkACOI` script will find the `acoi-config` script and use it to set a wide variety of environment variables. Some of these variables may be usefull to be set in your login script:

ACOI_MOD_PATH where to find Acoi plugins and detectors;

ACOIPATH where to find the feature grammar specifications, needed to include feature grammar modules.

When you set those variables yourself the `checkACOI` script will append the default settings at the end.

2.3 Installing Acoi for developers

Developers will create their feature grammars in different directories then the installed Acoi toolkit. Their basic setup is installed as follows:

— *command line*

```
1: (acoi 1) cd ~
2: (acoi 2) mkdir acoi.user
3: (acoi 3) cd acoi.user
4: (acoi 4) gtar xvfz ~/tmp/grammars.tgz
5: grammars/
6: ...
7: grammars/conf/start
```

—

Some additional environment variables are needed to be able to run the developed feature detector engine (FDE).

— *~/.bashrc*

```
1: ...
2: export ACOIHOME="$HOME/acoi.user/grammars/Linux"
3:
4: export PATH="$ACOIHOME/bin:$PATH"
5: ...
```

—

`ACOIHOME` tells where the Acoi user directory is.

The next chapter will guide you through the setup and construction of your first feature grammar.

Chapter 3

My first feature grammar

3.1 The feature grammar

This example feature grammar will extract the basic ID3 tags from local mp3 audio files. ID3 tags are glued to the end of a normal mp3 file and contain additional information, *e.g.* the title of the song.

Mx (see http://monetdb.cwi.nl/TechDocs/Services/Mx/index.html) is a tool used by MonetDB to combine documentation and source code, *e.g.* a C header and its implementation, into one file. The mp3 feature grammar part of the Mx file looks as follows:

— *~/acoi.user/grammars/mp3/mp3.mx*

```
 1: @f mp3
 2: @fgr
 3: %module          mp3;
 4:
 5: %start           mp3(location);
 6:
 7: %detector        ID3(ancestor::mp3/location);
 8:
 9: %atom   str      location,title,performer,album,genre;
10: %atom   int      year;
11:
12: mp3              : location ID3;
13: ID3              : title performer album year genre;
```

—

3.2 The detector implementation

There is only one detector in this grammar, `ID3`, and for this one we have to provide an implementation. We can generate a skeleton for this implementation using the `fgEngine` tool.

— *command line*

```
1: (acoi 1) cd ~/acoi.user/grammars/mp3
2: (acoi 2) Mx -c mp3.mx
3: mp3.mx: ./mp3.fgr - created
4: (acoi 3) fgEngine --skeleton -g ./mp3.fgr >> mp3.mx
5: (acoi 4) rm mp3.fgr
```

—

The skeleton code looks as follows (note the `@c` at line 2 manually added in the file to mark the output as C code).

— *~/acoi.user/grammars/mp3/mp3.mx*

```
 1: ...
 2: @c
 3: /* Copyright 2002 (C) Menzo Windhouwer, CWI */
 4:
 5: #include <fde/fd.h>
 6:
 7: float mp3_ID3_detector(tokenpool *tp,cursor *C0) {
 8:   float res = FAILED;
 9:   {/* Begin of the user-supplied code */
10:
11:   }/* End of the user-supplied code */
12:   return res;
13: }
```

—

The `mp3_ID3_detector` function takes two parameters:

`tokenpool *tp` The tokenpool as consumed by the FDE.

`cursor *C0` A cursor to the `location` node in the parse tree, *i.e.* the parameter of the detector declaration.

The detector returns a float. If you just want to indicate success or failure use the `SUCCESS` and `FAILED` defines. But you can also return a value between zero and one, thus indicating the *confidence* of the detector in its output. This confidence value appears as an attribute of the detector node in the parse tree.

At http://sourceforge.net/projects/id3lib/ we find a library which helps us to extract the ID3 tags. If you installed the library we can add the following check for it to `configure.ag`:

— *~/acoi.user/grammars/configure.ag*

```
 1: ...
 2: dnl check for ID3
 3: have_id3=auto
 4: ID3DIST=""
 5: ID3_INCS=""
 6: ID3_LIBS=""
 7: ID3_CFLAGS=""
 8: AC_ARG_WITH(id3,
 9: [  --with-id3=DIR     ID3 is installed in DIR], have_id3=
  "$withval")
10: if test "x$have_id3" != xno; then
11:   if test "x$have_id3" != xauto; then
12:     ID3DIST="$withval"
13:   else
14:     AC_CHECK_PROG(ID3INFO,id3info,id3info)
15:     if test "x$ID3INFO" != x; then
16:       ID3DIST=`type id3info|sed -e 's/^id3info is
  \(.*\)\/bin\/id3info$/\1/'`
17:       if test "x$ID3DIST" != x; then
18:         have_id3=yes
19:       fi
20:     fi
21:   fi
22:
23:   if test "x$have_id3" != xauto; then
24:     ID3_INCS="-I$ID3DIST/include"
25:     save_CPPFLAGS="$CPPFLAGS"
26:     CPPFLAGS="$CPPFLAGS $ID3_INCS"
27:     AC_CHECK_HEADER(id3.h, have_id3=yes, have_id3=no)
28:     CPPFLAGS="$save_CPPFLAGS"
29:   fi
30:
31:   if test "x$have_id3" = xyes; then
```

```
32:     ID3_CFLAGS=""
33:     ID3_LIBS="-L$ID3DIST/lib -lid3 -lz"
34:   else
35:     ID3DIST=""
36:     ID3_INCS=""
37:     ID3_LIBS=""
38:     ID3_CFLAGS=""
39:   fi
40: fi
41: AC_SUBST(ID3DIST)
42: AC_SUBST(ID3_INCS)
43: AC_SUBST(ID3_LIBS)
44: AC_SUBST(ID3_CFLAGS)
45: AM_CONDITIONAL(HAVE_ID3,test x$have_id3 = xyes)
46: ...
```

—

We try to find the library by searching for the `id3info` program. If this search is successful several variables are set, *i.e.* `ID3_INCS`, `ID3_LIBS` and `ID3_CFLAGS`. Later we can use these variables in our `Makefile.ag`. If you want some more information about what is happening here you can issue the command: `info autoconf`.

Using this libary we rewrite the `ID3Detector` skeleton in `mp3.mx`:

— *~/acoi.user/grammars/mp3/mp3.mx*

```
 1: @f mp3
 2: @fgr
 3: %module         mp3;
 4:
 5: %start          mp3(location);
 6:
 7: %detector       ID3(ancestor::mp3/location);
 8:
 9: %atom   str     location,title,performer,album,genre;
10: %atom   int     year;
11:
12: mp3             : location ID3;
13: ID3             : title performer album year genre;
14: @c
15: #include "fd.h"
16:
17: #include <id3.h>
```

```
18:
19: @= ID3Field
20:   if (my_frame=ID3Tag_FindFrameWithID(my_tag,@1)) {
21:     if (my_field=ID3Frame_GetField(my_frame,ID3FN_TEXT)) {
22:        ID3Field_GetASCII(my_field,buf,BUFSIZ);
23:        tp_mark_append(tp,"@2",buf,m);
24:     }
25:   }
26: @c
27:
28: float mp3_ID3_detector(tokenpool *tp, cursor *c_location) {
29:   float res = FAILED;
30:   ID3Tag *my_tag = NULL;
31:   ID3Frame *my_frame = NULL;
32:   ID3Field *my_field = NULL;
33:
34:   mark m = new_mark();
35:
36:   char buf[BUFSIZ];
37:
38:   if (my_tag = ID3Tag_New()) {
39:     char *loc = c_get_value(c_location);
40:     ID3Tag_Link(my_tag,loc);
41:     @:ID3Field(ID3FID_TITLE,title)@
42:     @:ID3Field(ID3FID_LEADARTIST,performer)@
43:     @:ID3Field(ID3FID_ALBUM,album)@
44:     @:ID3Field(ID3FID_YEAR,year)@
45:     @:ID3Field(ID3FID_CONTENTTYPE,genre)@
46:     res = SUCCESS;
47:     free(loc);
48:   }
49:
50:   return res;
51: }
```

—

A walkthrough will enlighten you about the changes:

line 28 Changed the non-descriptive `C0` variable name to `c_location`.

lines 30-32 These `my_*` variables are used by the ID3 library.

line 36 A buffer for intermediate results.

line 39-40 The function `c_get_value` gets the `location` of the `mp3` file. And this file is opened by the ID3 library. `c_get_value` returns a copy of the value, so we keep it in a variable to free it later (line 47).

lines 41-45 These lines contain calls to the Mx macro `ID3Field` which contains the ID3 library commands to extract a specific tag.

lines 19-26 A Mx macro can take up to nine parameters, which are refered to by `@1` to `@9`. The macro also contains the command to put the tag value into the tokenpool: `tp_mark_append`.

3.3 Compiling the FDE

What is left is to make a `Makefile.ag` to glue everything together into a dynamic library.

— *~/acoi.user/grammars/mp3/Makefile.ag*

```
 1: INCLUDES         = $(ID3_INCS)
 2: CFLAGS           = $(ACOI_CFLAGS) $(ID3_CFLAGS)
 3:
 4: lib__fd_mp3      = {
 5:          SOURCES = mp3.mx
 6:          LIBS    = $(ACOI_LIBS) $(ID3_LIBS)
 7: }
 8:
 9: headers_fgr      = {
10:          HEADERS = fgr
11:          DIR     = $(includedir)
12:          SOURCES = mp3.mx
13: }
```

—

`autogen` is a bit picky about the format of this file, *e.g.* make sure that there is whitespace around the '=' as otherwise it will not be able to parse it correctly.

To link this `Makefile.ag` into the compilation process we need to tell the top level `Makefile.ag` that we added the `mp3` subdirectory.

— *~/acoi.user/grammars/Makefile.ag*

```
1: SUBDIRS = conf HAVE_ID3?mp3
```

—

We made the `mp3` subdirectory optional in the compilation proces. It depends on the `HAVE_ID3` conditional as specified in the ID3 library check we added to `configure.ag`. So now the `mp3` FDE will only be compiled when the ID3 library is found and checked by `configure`.

The following commands compile the `mp3` FDE.

— *command line*

```
 1: (acoi 1) bootstrap
 2: /home/acoi/acoi.user/grammars/conf
 3: /home/acoi/acoi.user/grammars/mp3
 4: (acoi 2) mkdir Linux
 5: (acoi 3) cd Linux
 6: (acoi 4) ../configure --prefix=`pwd`
 7: creating cache ./config.cache
 8: ...
 9: creating fgr_config.h
10: (acoi 5) make
11: cd .. && autoheader
12: ...
13: gmake[1]: Leaving directory `/home/acoi/acoi.user
  /grammars/Linux'
14: (acoi 6) make install
15: Making install in conf
16: ...
17: gmake[1]: Leaving directory `/home/acoi/acoi.user
  /grammars/Linux'
```

—

In-depth

The `Makefile` calls the `fgEngine` tool to generate the FDE glue code: `fgEngine --glue -g mp3 > mp3_glue.c`. `fgEngine` uses the `ACOIPATH` to find `mp3.fgr`. `mp3_glue.c` is combined with `mp3.c` and compiled and linked into `lib_fd_mp3.so`. This library is placed in `$ACOIHOME/lib`, which directory we also have added to the `ACOI_MOD_PATH` search path, so it can be found. The feature grammar, `mp3.fgr`, is copied into `$ACOIHOME/include`.

3.4 Running the FDE

Now we can run the `mp3` FDE as follows:

— *command line*

```
 1: (acoi 1) fgEngine -g mp3 -L $ACOIHOME/lib
  location:/home/acoi/mp3/Anouk/Together_Alone/Mood_Indigo.mp3
 2: <?xml version="1.0"?>
 4: <mp3 ... >
 5:   <location ... >
 6:     <str ... >/home/acoi/mp3/Anouk/Together_Alone
  /Mood_Indigo.mp3</location>
 7:     </str>
 8:   </location>
 9:   <ID3 ... >
10:     <title ... >
11:       <str ... >Mood Indigo</str>
12:     </title>
13:     <performer ... >
14:       <str ... >Anouk</str>
15:     </performer>
16:     <album ... >
17:       <str ... >Together Alone</str>
18:     </album>
19:     <year ... >
20:       <int ... >0</int>
21:     </year>
22:     <genre ... >
23:       <str ... >(17)</str>
24:     </genre>
25:   </ID3>
26: </mp3>
```

—

The first line shows the FDE call. It takes the initial token, the `location` token, as one of the command line argument. The `-L` command line argument tells the FDE where to find the detector. Lines two to twentysix show the XML output, excluding all attributes, containing the parse tree. In fact the XML output is less

nicely formatted, as there is no additional whitespace added for pretty printing. Just redirect the XML output to a file, *e.g.* `~/tmp/out.xml`, and use `xmllint --format ~/tmp/out.xml` to get a nicer view (`xmllint` comes with the libxml package).

Getting more information for debugging purposes is done by calling the FDE with the `--debug` switch (the short version is `-d`) and/or the `--verbose` switch (:

— *command line*

```
1: (acoi 1) fgEngine --debug -g mp3 -L $ACOIHOME/lib
 location:/home/acoi/mp3/Anouk/Together_Alone/Mood_Indigo.mp3
2: ?DEBUG:BEGIN:validate start[mp3::mp3] tokenpool[location:
 /home/acoi/mp3/Anouk/Together_Alone/Mood_Indigo.mp3]
3: ...
4: ?DEBUG: END :validate start[mp3::mp3] result[SUCCESS]
 tokenpool[]
5: <?xml version="1.0"?>
6: ...
7: </mp3>
```

—

What is done with this XML document depends on the type of application. One could be insertion into a meta-index for later use by a search engine for the local mp3 files. Acoi provides tools to do this, and they will be described in the next chapter.

Chapter 4

My first meta-index

The mp3 feature grammar we constructed can only handle one file. In reality we want to index a collection of mp3 files and open them up for search. To enable this we will have to extend the grammar a bit:

— *~/acoi.user/grammars/mp3/mp3.mx*

```
 1: @f mp3
 2: @fgr
 3: %module          mp3;
 4:
 5: %start           collection(location);
 6:
 7: %detector        exec::select(preceding::location)
  = "mp3_select";
 8:
 9: %start           mp3(location);
10:
11: %detector        ID3(ancestor::mp3/location);
12:
13: %atom   str      location,title,performer,album,genre;
14: %atom   int      year;
15:
16: collection       : location select;
17: select           : &mp3*;
18: mp3              : location ID3;
19: ID3              : title performer album year genre;
```

—

In line 5 we add an extra start symbol, `collection`. This start symbol takes the root directory of the collection as a parameter. The `select` detector of

line 7, implemented by the `exec` plugin, finds the locations of all the mp3 files in this collection. The `exec` plugin executes an external program, in this case `mp3_select`:

— *~/acoi.user/grammars/mp3/mp3_select.mx.in*

```
1: @f mp3_select
2: @sh
3: #!@BASH@
4:
5: if [ ! -z $1 ]; then
6:          find $1 -name '*.mp3' -type f -print
 | awk '{ print "location:"$0 }'
7: fi
```

—

The `.in` extension means that the file will be parsed by `configure` and some variables will be replaced (*i.e.* `@BASH@`).

From time to time new mp3 files will be added to the collection. These changes will have to be reflected in the meta-index. To achieve this we will revalidate the collection once each day. This is done by specifying a lifespan (in milliseconds) for `mp3::collection`.

— *~/acoi.user/grammars/mp3/mp3.mx*

```
1: @f mp3
2: @fgr
3: ...
4: @mil
5: MEMO_symbol("mp3","collection",lng(24 * 60 * 60 * 1000));
```

—

This MIL script is loaded into MonetDB and the lifespan is used when the FDE/FDS asks for a new candidate.

4.1 Creating the meta-index

The first step is to setup some environment variables, which we will put it in a small script so we can easily source it.

— *~/acoi.user/grammars/mp3/setenv.sh*

```
 1: export ACOIFDE="mp3"
 2: export ACOIDB="mp3"
 3: export MAPIPORT="60010"
 4: export ZEEPPORT="60020"
 5: export ZEEPHOST=`hostname`
 6: export ACOIPORT="60030"
 7: export ACOIHOST=`hostname`
 8: export ACOITMP="$ACOIHOME/$ACOIFDE/tmp"
 9: export ACOILOG="$ACOIHOME/$ACOIFDE/log"
10: export ACOISTART="-s mp3::collection"
11: export ACOIINIT="location:/home/acoi/mp3"
```

—

line 1 `ACOIFDE` is the only mandatory one and, as the name already suggests, specifies the FDE name.

line 2 By default `ACOIDB` is the same as `ACOIFDE`, but if you want to store the meta-data in a different database change this setting.

line 3 to 7 Change these when the database server (`MAPI` and `ZEEP`) and the FDS (`ACOI`) run on different machines then the FDE and listen to non-default ports.

line 8 and 9 The places for intermediate (`ACOITMP`) and log files (`ACOILOG`).

line 10 During parsing only one start symbol can be the root of the tree. If there are multiple start symbols, *e.g.* `mp3::collection` and `mp3::mp3`, the `ACOISTART` variable is used to specify which start symbol to take.

line 11 The initial token, *i.e.* the root of the mp3 collection.

Now we can initialize the database:

— *command line*

```
 1: (acoi 1) cd ~/acoi.user/grammars/mp3/
 2: (acoi 2) source setenv.sh
 3: (acoi 3) initACOI
 4: !WARNING: no ACOIDBMS specified, default monet used
 5: !WARNING: no ACOIMODE specified, default shallow used
 6: !WARNING: no ACOIWAIT specified, default forever used
 7: ?LOG: /home/acoi/acoi.user/userguide.ID3/Linux/mp3/tmp
  dir created
```

```
 8: ?LOG: /home/acoi/acoi.user/userguide.ID3/Linux/mp3/log
  dir created
 9: ?LOG: /home/acoi/acoi.user/userguide.ID3/Linux/mp3/log
  /error dir created
10: !WARNING: GDKlockHome: ignoring empty or invalid .gdk_lock.
11: !WARNING: BBPdir: initializing BBP.
```

—

`initACOI` checks the environment variables, creates, if needed, the directories and calls the `Mserver` for the first time, which will initialize the database.

In-depth

For the database creation script the feature grammar is first translated into an XML document using: `fgEngine --schema`. A DBMS specific XSLT script (*i.e.* `monet/schema.xsl`) is then used to translate this into the DBMS specific schema creation statements.

In the case of MonetDB the database schema is created the first time the server is started.

Future

`fgEngine --schema` generates a feature grammar specific schema format. This should be replaced by a standard like XML Schema or Relax NG, so it can also be used to validate the results of a FDE.

To install these new files we have to add them to the `Makefile.ag`.

— *~/acoi.user/grammars/mp3/Makefile.ag*

```
1: mp3libdir  = libdir/$(subdir)
2:
3: ...
4:
```

```
 5: scripts_sh = {
 6:     DIR     = mp3libdir
 7:     SOURCES = mp3_select.mx.in
 8: }
 9:
10: scripts_mil = {
11:     DIR     = mp3libdir
12:     SOURCES = mp3.mx
13: }
```

—

Now we have to reinstall the mp3 subdirectoy.

— *command line*

```
 1: (acoi 1) cd ~/acoi.user/grammars
 2: (acoi 2) bootstrap
 3: /home/acoi/acoi.user/grammars/conf
 4: /home/acoi/acoi.user/grammars/mp3
 5: (acoi 3)
 6: (acoi 4) cd Linux
 7: (acoi 5) ../configure  --prefix=`pwd`
 8: loading cache ./config.cache
 9: ...
10: fgr_config.h is unchanged
11: (acoi 6) cd mp3
12: (acoi 7) make
13: /home/acoi/monet/Linux/bin/Mx -n -x c ../../mp3
  /mp3.mx ../../mp3/mp3.mx: ./mp3.c - created
14: ...
15: gcc -I /home/acoi/acoi/Linux/include/fg -o mp3 mp3.o
  mp3_engine.o -L/home/acoi/acoi/Linux/lib -lfg -lfde
  -L/home/acoi/local/Linux/lib -lid3 -lz
16: (acoi 8) make install
17: gmake[1]: Entering directory `/home/acoi/acoi.user
  /grammars/Linux/mp3'
18: ...
19: gmake[1]: Leaving directory `/home/acoi/acoi.user
  /grammars/Linux/mp3'
```

—

We have to do the `bootstrap` again as a `Makefile.ag` has been changed. Furthermore, we added some `.in` files which have to be expanded, and this is done in the `configure` run.

We are now ready to really start the engine.

4.2 Populating the meta-index

The first step is starting the database server:

— *command line*

```
1: (acoi 1) startACOI
2: starting ACOI(mp3) main services ...
3:     starting ACOI(mp3) DBMS backend ...
4:         starting MONET server ...
5:         ... done
6:     ... done
7:     starting ACOI(mp3) FDS ...
8:     ... done
9: ... done
```

—

Now we can also start the robot:

— *command line*

```
1: (acoi 1) startACOIrobot
2: starting ACOI(mp3) robot ...
3:     starting ACOI(mp3[fde]) clients ...
4:         starting ACOI(mp3[fde]) FDE ...
5:         ... done
6:     ... done
7: ... done
```

—

When the candidate pool is empty the robot will wait forever (see $ACOI-WAIT) for new candidates. After the specified lifespan the `mp3::collection` will become valid and the collection will be scanned for new mp3 files.

The following commands stop both the robot and the server:

— *command line*

```
 1: (acoi 1) stopACOIrobot
 2: stopping ACOI(mp3) robot ...
 3:     stopping ACOI(mp3) clients ...
 4:         stopping ACOI(mp3[fde]) FDE ...
 5:         ... done
 6:     ... done
 7: ... done
 8: (acoi 2) stopACOI
 9: stopping ACOI(mp3) main services ...
10:     stopping ACOI(mp3) FDS ...
11:     ... done
12:     stopping ACOI(mp3) DBMS backend ...
13:         stopping MONET server ...
14:             commit ...
15:             ... done
16:             shutdown ...
17:             ... done
18:         ... done
19:     ... done
20: ... done
```

—

When something goes wrong information can be found in the following files:

$ACOITMP/ERROR.mp3 Something went really wrong. The message will have been shown on your screen, but is also stored in this file. A serious error can occur when the robot can not find one of the scripts used as hook.

$ACOITMP/WARNING.mp3 Non life threatening messages are stored in this file, and they may contain hints for a latter crash.

$ACOILOG/mp3.monet.log The log messages from MonetDB. There is lots of information here. The log messages have the following shared format:

```
[ 4/ 3@Mon Jul  9 15:29:30 2001]DONE :add:mp3::collection(...)
```

The first number is the clientid while the second one is the sessionid, these are needed for debugging concurrent executed scripts. The part after the '@' is the date/time, but I guess you figured that out. After this header follows the action information. The first uppercase word indicate the status of the action, which is typed in lower letters, followed by some free text. So

this message indicates the end of an “add” action of a successfull explored candidate.

If MonetDB encounters a serious problem you will also find it here.

$ACOILOG/mp3.acoi.fde.log and $ACOILOG/mp3.acoi.fds.log The log messages from the FDE and FDS. You can get a lot of information by setting $ACOIARGS to `"-v -d"`.

4.3 Querying the meta-index

There is some rudimentary support for querying the meta-index. If you want rather advanced queries you will have to write your own MIL scripts and directly access the MonetDB tables.

The following commands start MonetDB, the query server and the query GUI:

— *command line*

```
1: (acoi 1) startACOI
2: starting ACOI(mp3) main services ...
3: ...
4: ... done
5: (acoi 2) java acoi.query.server &
6: [...]DONE :loaded file:/home/acoi/acoi/Linux/lib/Acoi
 /monet/query.xsl
7: [...]BEGIN:listening at 60040
8: (acoi 3) java acoi.query.main
 /home/acoi/acoi.user/grammars/Linux/lib/mp3/mp3.xml
```

—

The example query in Figure 4.1 asks for all the titles from mp3 files with a performer called “Anouk”.

In-depth

The query interface generates an XML representation of the query.

—

```
1: <?xml version="1.0"?>
2: <fgquery id="0" grammar="/.../mp3.xml" start="mp3">
```

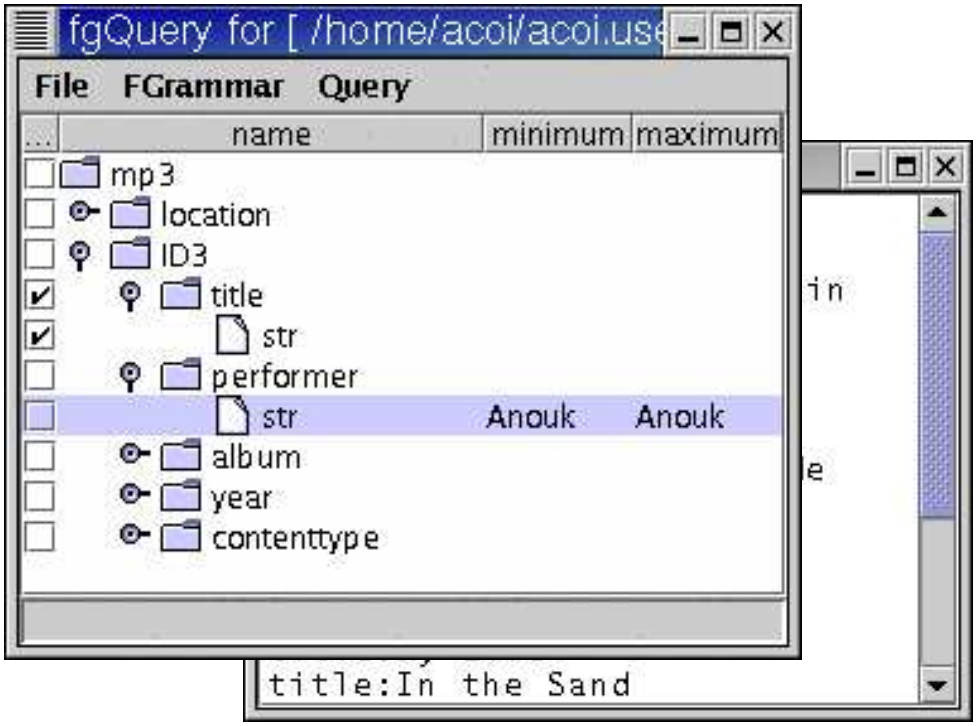

Figure 4.1: Example query and answer

```
 3:   <select id="1">
 4:     <and id="2">
 5:       <mp3  id="3"  type=".grammar.non-terminal.start."
  modes=".detect.store.query.">
 6:         <ID3  id="4"  type=".grammar.non-terminal.detector.
  blackbox." modes=".detect.store.query.">
 7:           <performer  id="5"  type=".grammar.non-terminal."
  modes=".detect.store.query.">
 8:             <str  id="6"  type=".grammar.terminal.atom."
  modes=".detect.store.query." minimum="Anouk"
  maximum="Anouk">
 9:             </str>
10:           </performer>
11:         </ID3>
12:       </mp3>
13:     </and>
14:   </select>
15:   <project id="7">
16:     <mp3  id="8"  type=".grammar.non-terminal.start."
```

```
  modes=".detect.store.query.">
17:       <ID3  id="9"  type=".grammar.non-terminal.detector.
  blackbox." modes=".detect.store.query.">
17:         <title  id="10"  type=".grammar.non-terminal."
  modes=".detect.store.query." project="true">
18:           <str  id="11"  type=".grammar.terminal.atom"
  modes=".detect.store.query." project="true">
19:           </str>
20:         </title>
21:       </ID3>
22:     </mp3>
23:   </project>
24: </fgquery>
```

—

This XML document consists of two parts. The first part contains a part of feature grammar parse tree and describes all the selection criteria. The next part is again a parse tree, but describes now projection flags. These two parts can describe simple SJP (selection, join, projection) queries, where the join part is implicitly described by the parse tree.

The query server recieves this XML document and expands it with the `monet/query.xsl` templates into a native DBMS query, *i.e.* a MIL script. The DBMS should reply with an XML document, which, optionally, may run against another XSLT script to translate the answer. The query interface uses `tree.xsl` to translate the XML answer into a HTML page.

Chapter 5

Is that all there is?

Of course not! In this chapter you will find some more information on supported feature grammar language functionality, detector implementation and the robot infrastructure. Only when it is beneficial to the functionality of the mp3 FDE we will extend the example further.

5.1 Feature grammar language

The mp3 feature grammar is quite basic. The language has some more functionality which can be used. In this section we will go over an EBNF specification of the feature grammar language. The EBNF notation is the same as used in the XML standard (see http://www.w3.org/TR/1998/REC-xml-19980210). Furthermore some parts directly relate to the XPath 1.0 specification (see http://www.w3.org/TR/xpath), and a clear understanding of it will help understanding their adoption in the Acoi system.

— *feature grammar EBNF*

```
1: digit          ::= [0-9]
2: exponent       ::= [Ee][-+]?{D}+
3: letter         ::= [_a-zA-Z]
4: any            ::= [#x0-#xFFFE]
5: symbol         ::= letter ( letter | digit )*
6: scope          ::= letter ( letter | digit )*
7: prefix         ::= scope "::"
```

—

First some basic types which we will encounter throughout the language. The `symbol` names the most basic building blocks of a feature grammar the terminals

and non-terminals. Those symbols may have to be bound to a certain `scope`, using the `prefix` notation. The `scope` may refer to different things: a feature grammar module, an ADT module or a detector plugin.

— *feature grammar EBNF*

```
 8: feature-grammar::= module-decl decl*
 9: decl            ::= use-decl | start-decl | atom-decl |
                        detector-decl | hook-decl | mode-decl |
                        version-decl | rule-decl
```

—

The first statement is a feature grammar is a module declaration, as this is needed to bind all new symbols to the correct scope. The module declaration is followed the other declarations in arbitrary order.

— *feature grammar EBNF*

```
10: module-decl    ::= "%module" scope ';'
11: use-decl       ::= "%use" scope ';'
```

—

The module name should be equal to the filename, this to enable the reusage of feature grammar modules using the use declaration. The use declaration is skipped when the module is already loaded. Feature grammar module specifications are searched for using the `ACOIPATH` environment variable or the search path passed on to `fgEngine` using the `--include=<paths>` command line option.

— *feature grammar EBNF*

```
12: start-decl     ::= "%start" symbol forward-params ';'
13: forward-params ::= '(' forward-paths? ')'
14: forward-paths  ::= forward-path ( ',' forward-path )*
15: forward-path   ::= symbol ( '/' symbol )*
```

—

The start symbol takes a set of paths as parameters. These paths point to descendants of the symbol, and can be used to resolve references. They also indicate which tokens should be (minimally) available in the input sentence. Currently no XPath axes are allowed in these paths.

— *feature grammar EBNF*

```
16: atom-decl       ::= "%atom" prefix? symbol
                        ( ( symbol ( ',' symbol )* )? |
                          '{' any+ '}' ) ';'
```

—

The atom declaration adds support for a new ADT, where the prefix indicates which ADT extension module the DBMS should load to support this ADT. If the prefix isn't explicitly given the name of the ADT module is assumed to be the same as the atom symbol name. When the atom is already known there is no new atom added to the internal symbol table. Most of MonetDBs builting atoms are by default supported, *i.e.* `bit`, `chr`, `sht`, `int`, `oid`, `flt`, `dbl`, `lng` and `str`. The declaration may be followed by a list of symbols. Those symbols should be non-terminals and an atom rule is added for them. In fact this declaration is a shortcut for such a rule, *e.g.* `%atom str title,body;` is the equavalent to the following two rules: `title: str; body: str;`. The declaration of a new atom may also be followed by a regular expression (embedded in curly brackets). This regular expression is then used to validate the tokens which are consumed during the parsing process. Using this mechanism runtime errors when invalid data is inserted into the database may be prevented. For example: `%atom image::bitmap {^(0|1)*$};` would make sure that all values of a bitmap atom contain only zeros and/or ones. The regular expression should conform to the POSIX Extended Regular Expression syntax, which is a bit explained by Jan Wolter (see http://www.unixpapa.com/incnote/regex.html): *Basic regular expressions are similar to those in* `ed`*, while extended regular expressions are more like those in* `egrep`*, adding the '|', '+' and '?' operators and not requiring backslashes on parenthesized subexpressions or curly-bracketed bounds.* The complete specification of the basic and extended syntax is found at several places on the web, for example: http://sunland.gsfc.nasa.gov/info/regex/Top.html. The validation of the builtin (MonetDB) types is not too extensive, *e.g.* `sht` and `int` share the same regexp, so there is no check on the valid range.

— *feature grammar EBNF*

```
17: detector-decl  ::= "%detector" prefix? symbol detector-params ';'
18: detector-params::= '(' ( detector-param ( ',' detector-param )* )? ')'
19: detector-param ::= string | xpath
20: xpath          ::= absolute | relative
21: absolute       ::= '/' relative? | "//" relative
22: relative       ::= step ( '/' step | "//" step )*
23: step           ::= axis? ( ( symbol | '*' ) | dereference ) |
```

```
                        abbreviation
24: axis           ::= "self::" | "parent::" | "child::" |
                       "ancestor::" | "ancestor-or-self::" |
                       "preceding::" | "preceding-sibling::"
                       "descendant::" | "descendant-or-self::" |
                       "following::" | "following-sibling::"
25: abbreviation   ::= '.' | ".."
26: dereference    ::= '&' symbol
```

—

The declaration of a blackbox detector takes a set of paths refering backwards into the parse tree as parameters. These paths conform to a subset of the XPath specification, most notably they don't allow step qualifiers. A detector may be bound to a plugin using the `prefix`, this plugin will be loaded using the `ACOI_MOD_PATH` environment variable and will interpret the symbol and its context. When there is no plugin specification the developer will have to provide the implementation.

— *feature grammar EBNF*

```
27: detector-decl  ::= "%detector" prefix? symbol
                       '[' any+ ']' ';'
```

—

This detector declaration defines a whitebox detector. The implementation of the whitebox is embedded between square brackets. When you need square brackets in the implementation escape them with a '\', *e.g.* `%detector isFoo [ preceding::* \[ text = "foo" \] ];` The feature grammar language doesn't describe the exact nature of this implementation, it's interpreted by the plugin associated with the whitebox detector. By default, *i.e.* when there is no plugin specified, this is the `xpath` plugin. This plugin expects a valid XPath step qualifier which it will evaluate in the context of the detector node. When you enter an invalid expression you won't get any message during parsing of the feature grammar but a runtime error will be raised during feature detection.

— *feature grammar EBNF*

```
28: hook-decl      ::= "%detector" symbol '.'
                       ( "init" | "final" ) "()" ';'
```

—

A very limited set of hooks allow the developer to insert own code at specific points in the parser. The `init` hook is executed the first time a symbol is encountered during feature detection. Which makes this hook ideal to initialize a library or to setup a socket connection. The `final` hook is executed when feature detection is finished, but only when an accompanying `init` hook has also been executed.

—*feature grammar EBNF*

```
29: mode-decl      ::= '%' ( "transient" | "query" ) symbol ';'
```

—

By default all information is available during the three modes of the system: detection, persistent storage and querying. Using the mode declarations the availability of partial parse trees may be limited to a subset of these modes. The `transient` mode is used to limit a symbol (and its descendants) to the detection phase. This maybe used for temporary files, *e.g.* the local place of a cached web object. The information will not be stored and thus is also not available for queries. On the other side is the `query` mode: in this case symbols are only available for querying. This is usefull to include a "static" set of database tables in the query process, *e.g.* tables containing the synsets of WordNet. For all modes the symbol, which is the root of the partial parse tree for which this mode will hold, should be a detector, so the "missing" information (as it isn't persistently stored) can be (re)produced just-in-time.

—*feature grammar EBNF*

```
30: version-decl   ::= "%version" symbol unsigned-integer '.'
                       unsigned-integer '.' unsigned-integer ';'
```

—

Detectors maybe tagged with a version. The version consists of three revision levels: major, minor and correction. This version is inserted as an attribute of the detector node into the parse tree, and may thus be stored in the database. The `fgSchedular` should use this information to invalidate and revalidate the output of this detector when the version, and thus the implementation, of the detector changes.

—*feature grammar EBNF*

```
31: rule-decl      ::= prefix? symbol ':' lhs ';'
```

```
32: lhs             ::= ( prefix? symbol bounds? | constant )+
33: lhs             ::= '(' lhs ')'
34: lhs             ::= lhs '|' lhs
35: bounds          ::= list | set | tuple
36: list            ::= '[' range ']' | range
37: set             ::= '{' range '}'
38: tuple           ::= '<' int-range '>'
39: range           ::= wild-range | int-range
40: int-range       ::= unsigned-integer ( ':' unsigned-integer )?
41: wild-range      ::= '*' | '+' | '?'
42: constant        ::= float | integer | string
43: float           ::= '-'? digit+ '.' digit+ exponent?
44: integer         ::= '-'? digit+
45: unsigned-integer::= digit+
46: string          ::= '"' any* '"'
```

—

Finally we reached the feature grammar rules. Normally you will add rules for non-terminals in the scope of the current module. However, you may also add alternative rules for symbols from imported modules, but you can't add new non-terminals to the scope of these imported modules. The right-hand side of the rule consists of a list of symbols. For each symbol a module scope may appear as prefix, use this to resolve naming conflicts. Symbols may be repatitive, which is indicated by the bounds. The repetition is embedded in a collection type: either a list (also the default), a set or a tuple. When the range of the bound isn't fixed wildcards can be used. Finally constants may also appear on the right-hand side of a rule.

When a feature grammar is finished `fgEngine --check` can be used to check the grammar for validity. The output of this check will inform you of constructs that are (possibly) hard or even impossible to parse by the `fgEngine`. A feature grammar is basically a context-free grammar and the `fgEngine` implements a recursive descent parser for those grammars. This type of parser can handle a large subset of all possible context-free grammars, however, it can't handle left-recursion. Fortunately there are well known ways to rewrite your grammar in such a way rhat left-recursion is eliminated (see for example the Red Dragon book: *Compilers - principles, techniques and tools* by Aho, Seti and Ullman). But this elimination can't be done automatic, so you as a developer will have to decide what to do about it.

After specifying the feature grammar the detectors, when they are not associated with a plugin, have to be implemented. The following sections describe the facilities for the implementation of detectors, hooks and plugins.

5.2 Detector implementation

As we already have seen in the mp3 example the `fgEngine` tool can generate a skeleton implementation for a blackbox detector. The remainder of this section tells you how to interact with the FDE.

5.2.1 Parse tree cursors

Parameters of a detector are passed on as cursors into the parse tree. The following cursor functions allow the access of the valid nodes in the parse tree:

`int c_has_result(cursor *c)` Returns `true` when the cursor points at a valid node.

`int c_cnt_results(cursor *c)` Returns the number of valid nodes.

`tree *c_get_result(cursor *c)` Returns the valid node or `NULL` if there isn't one.

`tree *c_next_result(cursor *c)` Get the next node which matches the path expression.

`char *c_get_name(cursor *c)` Get the name of the current node.

`char *c_get_value(cursor *c)` Return a copy of the value of the valid node. Remember that it's up to the caller to free this copy!

`char *c_get_attr(cursor *c,char *attr)` Return a copy of the value of the attribute of the valid node.

`void c_reset(cursor *c)` Reset the cursor to the first valid node.

5.2.2 Tokens and the tokenpool

This is the data structure which really is the domain of a feature detector. The tokenpool can be accessed with the following commands:

`int tp_is_empty(tokenpool *tp)` Returns `true` when the tokenpool does not contain any tokens.

`int tp_is_next(tokenpool *tp, char *name)` Returns `true` when the next token in the tokenpool has the specified name.

`token *tp_get(tokenpool *tp)` Get the next token.

`void tp_put(tokenpool *tp,char *name,char *value)` Put a token with this name and value in the front of the tokenpool.

`mark new_mark()` Create a new mark. In reality a mark is just a pointer to a token. The `new_mark` command initializes this as a `NULL` pointer, which is interpreted as the front of the tokenpool.

`void tp_mark_append(tokenpool *tp,char *name,char *value,mark m)` A new token is added after the mark, after which the mark is updated to this just inserted token.

Future

tp_put can not handle `NULL` values in the name or value. Values can now only be strings, this should become arbitrary pointers which can be accessed with `toXML` and `fromXML` functions. This way blob like structures like images could be embedded in the resulting XML document.

Future

Blackbox detectors should limit themselves to putting tokens into the tokenpool. By getting tokens from the tokenpool they create implicit data dependencies which can thus not be exploited by the upcomming scheduler. To enforce this the access to the global tokenpool may become more limited in the future, as has happened with the parse tree.

5.2.3 Properties

Some detector parameters do not belong in the parse tree and are just temporary adjustment for a specific run. These kind of properties can be specified on the command line:

— *command line*

```
1: (acoi 1) fgEngine --property videohost:medusa ...
```

—

This property can then be accessed as follows:

— *video.mx*

```
1: ...
2: int VIDEOsocket = SERVEopen(
3:   get_prop_or("videohost","utip053.cs.utwente.nl"),
4:   atoi(get_prop_or("videoport","3366")));
5: ...
```

—

The property functions are the following:

`void set_prop(char *name, void *value)` Set the property value. The name is copied, so it maybe a local variable. The value should be global, and maybe anything. It's up to you to free it.

`void *get_prop(char *name)` Get the property value, returns `NULL` if the property does not exist.

`void *get_prop_or(char *name,void *default)` Get the property value, or return the default value when the property is not set.

`int has_prop(char *name)` Return `true` when the property exists.

5.2.4 From log messages to fatal errors

The following commands can be used to give increasingly more urgent messages to the user:

`void fg_log(char *msg,...)` Print a log message when the verbose flag is on. The messages are printed with the `?LOG:` prefix.

`void fg_debug(int flag,char *msg,...)` Only prints this debug message when a specific debug flag is set. Use the `DEBUG_ALL` flag. These messages are printed with the `?DEBUG:` prefix.

`void fg_warning(char *msg,...)` Print a warning, prefixed by `!WARNING:`.

`void fg_error(char *msg,...)` Print an error message, which is prefixed by `!ERROR:`.

`void fg_fatal(char *msg,...)` A fatal error occured, prints the message prefixed by `!FATAL ERROR:` and then exits the program.

All these messages are printed on `stderr` so they do not interfere with the XML output.

5.3 Hook implementation

Hooks are rather simple as they are not allowed to have any influence on the feature detection process. The mp3 example doesn't contain any hooks, but lets create one.

—

```
1: %detector mp3.init();
```

—

The implementation could look as follows.

—

```
1: void mp3_mp3_init_hook() {
2:   fg_log("Welcome to the mp3 FDE!");
3: }
```

—

The FDE functionality, except for cursors and tokenpool access, described in the previous section, are of course also available for hooks and plugins (as described in the next section).

5.4 Plugin implementation

Implementing a plugin requires access to the heart of `fgEngine`, the symbol table, and thus should only be limited to experts of the Acoi API. However, you are an early adapator, so likely to become an expert! But then you should also be able to find your way around in the Acoi source tree. Here are some hints:

- `acoi/src/plugins/` is the directory containing the source code of the plugins comming by default with Acoi;
- `acoi/src/fg/symboltable.mx` and `acoi/src/fg/symbol.mx` contain the specifications of the symbols in the symbol table of the feature grammar. The plugin recieves the detector symbol, by analyzing the context of this symbol using these specifications it has to determine what to do.

5.5 Robot infrastructure

The robot infrastructure to built and populate the meta-index contains many hooks, both in the form of shell scripts and XSLT templates.

5.5.1 Overloading XSLT templates

schema.xsl Generate a script with the statements to create the database schema.

del.xsl Generate statements to delete parse trees from the database, this script should be set based, *i.e.* you can delete whole sets of documents in one go.

add.xsl The statements to add one parse tree to the database.

The first two XSLT scripts are based on the output of the `fgXML` tool. While the `add.xsl` templates expand the output of a FDE.

Both these types of XML documents contain generic attributes which are used for matching the templates.

— *command line*

```
1: (acoi 1) fgEngine --schema -g mp3
2: <?xml version="1.0"?>
3: <fgrammar module="mp3" name="mp3" modules="">
4:   <mp3 module="mp3" type=".grammar.non-terminal.start."
 modes=".detect.store.query.">
5:     <location module="mp3" type=".grammar.non-terminal."
 modes=".detect.store.query." coll="list" lbnd="1" hbnd="1"
 alt="0"/>
6:     <ID3 module="mp3"
 type=".grammar.non-terminal.detector.blackbox."
 modes=".detect.store.query." coll="list" lbnd="0" hbnd="1"
 alt="0"/>
7:   </mp3>
8: ...
```

```
 9:   <location module="mp3" type=".grammar.non-terminal."
  modes=".detect.store.query.">
16:     <str type=".grammar.terminal.atom." modes=".detect.store
  .query." coll="list" lbnd="1" hbnd="1" alt="0"/>
17:   </location>
18: ...
19: </fgrammar>
```

—

The generic attributes are the following:

type This attribute contains a dot seperated list describing the types the node belongs to.

modes The modes in which the node is valid.

coll The collection type of the node. Valid values are: `set`, `list`, `tuple`.

lbnd and hbnd The low and high bounds of a collection.

alt A non-terminal may have several alternative rules. This attribute indicates to which alternative a symbol belongs.

For example the following template from `add.xsl` generates MIL code to insert persistent atoms.

— *~/acoi/Linux/lib/Acoi/monet/add.xsl*

```
 1: ...
 2: <xsl:template match='*[contains(string(@type),".atom.")
  and contains(string(@modes),".store.")]' priority="0"
  mode="default_post">
 3:   <xsl:variable name="bat">
  <xsl:value-of select="name(parent::*)"/>_
  <xsl:value-of select="name()"/></xsl:variable>
 4:   <xsl:variable name="tail">
  <xsl:value-of select="name()"/>("
  <xsl:value-of select="."/>")</xsl:variable>
 5:
 6:   <xsl:call-template name="insert">
 7:     <xsl:with-param name="bat" select="$bat"/>
 8:     <xsl:with-param name="tail" select="$tail"/>
 9:   </xsl:call-template>
10: </xsl:template>
11: ...
```

—

If you want to change this default behaviour you will have to write a template which overloads specific templates.

Chapter 6

Detector toolbox

In this chapter some utilities are described which do not belong to the core of Acoi, however, they make implementing detectors more easy.

6.1 The serve library

Executables and libraries to be used may come in all kind of formats, which may not fit the format of the compiled FDE. For example a `n64` compiled FDE on `IRIX64` may not link in libraries compiled with `o32` or `n32`. To circumvent this we can wrap the library into a server compiled with the right format. The FDE can then communicate with this server. The Acoi distribution comes with a library which makes constructing such a server simple.

The `serve` library has the following interface for the server:

`int SERVEloop(char *serve,int port,SERVEhandler handler)`
Starts a service (called `serve`) on the specified port. Each time the server recieves a command it is passed on to the `handler`.

`int (*SERVEhandler)(char *command, char *reply)` The `handler` has to be implemented by the developer. He or she has to implement the actions to fill the reply buffer and return a status flag. The reactions on the status are as follows:

0 the server does nothing but synchronizing with the client;

1 the reply is sent to the client;

2 the client connection is closed;

3 the server is shut down.

Notice that each higher status also implies all the lower server actions.

`int SERVEmode(int mode)` The `mode` of the server is either `PROMPT` or `NEWLINE`, which determines how commands and replies are seperated.

`SERVE_DEBUG` **and** `SERVE_VERBOSE` Set these to `TRUE` to enable a level of additional information.

`void SERVEdebug(char *msg,...)` Prints debug messages when `SERVE_DEBUG` is `TRUE`.

`void SERVEverbose(char *msg,...)` Prints messages when `SERVE_VERBOSE` is `TRUE`.

`void SERVEwarning(char *msg,...)`

`void SERVEsyserror(char *msg,...)`

`void SERVEerror(char *msg,...)` Prints messages of increasing severity, where `SERVEerror` exits the program.

Depending on the mode the client communicates with the server using the following commands:

`int SERVEopen(char *host, int port)` Open a connection to a service.

`int SERVEclose(int sock)` Close the connection.

`int SERVEputs(int fd,char *buf)` Write a newline terminated command to the open connection.

`int SERVEgets(int fd,char *buf,int len)` Read a line from the open connection. When a complete line is read 1 is returned, otherwise a 0 and in the case of an error a -1.

`int SERVEputp(int fd,char *buf)` Write a prompt terminated command to the open connection.

`int SERVEgetp(int fd,char *buf,int len,int newline)` Read a prompt terminated reply from the open connection. When a complete reply is read 1 is returned, otherwise a 0 and in the case of an error a -1. The `newline` flag determines if the function also returns a 0 when a complete line is read.

The following example uses this library to execute a face detector which is only available for IRIX (see http://www.wi.leidenuniv.nl/~mlew/detect_face3.gz.bin).

— *serveFACE.mx.in*

```
 1: @f serveFACE
 2: @c
 3:
 4: #include <stdio.h>
 5: #include <stdlib.h>
 6: #include <string.h>
 7:
 8: #include <utilSERVE.h>
 9: #include <connSERVE.h>
10:
11: #define STREQ(a, b)  (strcmp(a, b)==0)
12:
13: static int port = 4488;
14:
15: static void usage(char* progname) {
16:   fprintf(stderr, "Usage: %s [<options>]\n",progname);
17:   fprintf(stderr, "\tOptions:\n");
18:   fprintf(stderr, "\t\t-h(elp)\n\n");
19:   fprintf(stderr, "\t\t-d(ebug)\n\n");
20:   fprintf(stderr, "\t\t-v(erbose)\n\n");
21:   fprintf(stderr, "\t\t-p(ort) port -- port to listen to\n\n");
22:   exit(1);
23: }
24:
25: static void handleArgs(int argc, char *argv[]) {
26:   int i = 0;
27:   for (i=1; i<argc; i++) {
28:     if ( (*argv[i] == '-')) {
29:       if (STREQ(argv[i], "-h") || STREQ(argv[i],"-host")) {
30:         usage(argv[0]);
31:       } else if (STREQ(argv[i], "-d") || STREQ(argv[i], "-debug")) {
32:         SERVE_DEBUG = TRUE;
33:       } else if (STREQ(argv[i], "-v") || STREQ(argv[i], "-verbose")) {
34:         SERVE_VERBOSE = TRUE;
35:       } else if (STREQ(argv[i], "-p") || STREQ(argv[i], "-port")) {
36:         if (i+1 >= argc)
37:           usage(argv[0]);
38:           if (atoi(argv[i+1]))
```

```
39:             port = atoi(argv[i+1]);
40:             i++;
41:       }
42:     } else
43:       usage(argv[0]);
44:   }
45: }
46:
47: void FACE_file(char *filename,char *reply) {
48:   FILE *pipe = NULL;
49:   char buf[BUFSIZ];
50:   strcpy(reply,"");
51:   sprintf(buf,"@FGR_PREFIX@/bin/ff %s",filename);
52:   pipe = popen(buf,"r");
53:   if (!pipe) {
54:     SERVEerror("couldn't open pipe to %s\n",buf);
55:     sprintf(reply,"error:couldn't open pipe to %s\n",buf);
56:     return;
57:   }
58:   while(fgets(buf,BUFSIZ,pipe) != NULL) {
59:     if (buf[0] == '!') {
60:       SERVEsyserror(buf);
61:       sprintf(reply,"%serror:%s\n",reply,buf);
62:     } else
63:       sprintf(reply,"%sface:%s\n",reply,buf);
64:   }
65:   pclose(pipe);
66: }
67:
68: int FACE_handler(char *command, char *reply) {
69:   if (strncmp(command,"stop",4) == 0) {
70:     return 3;
71:   } else if (strncmp(command,"quit",4) == 0) {
72:     return 2;
73:   } else if (strncmp(command,"file:",5) == 0) {
74:     char *filename = command+5;
75:     FACE_file(filename,reply);
76:     return 1;
77:   }
78:   return 0;
79: }
80:
81: int main(int argc, char *argv[]) {
```

```
82:   handleArgs(argc,argv);
83:
84:   SERVEverbose("serveFACE version 0.1\n");
85:   SERVEverbose("Copyright (c) 2001 CWI\n");
86:   SERVEverbose("\n");
87:
88:   SERVEmode(PROMPT);
89:
90:   SERVEloop("FACE",port,FACE_handler);
91:
92:   return 0;
93: }
```

—

The `IRIX` executable is wrapped in the following shell script:

— *ff.in*

```
 1: #!/bin/bash
 2:
 3: TMPDIR=$ACOITMP
 4:
 5: _PWD=`pwd`
 6:
 7: usage () {
 8:   echo "$0 <image file>"
 9:   exit 1
10: }
11:
12: if [ $# -ne 1 ]; then
13:   usage
14: fi
15:
16: SRC=$1
17:
18: if [ ! -f $SRC ]; then
19:   echo "!ERROR: $SRC doesn't exist"
20:   exit 1
21: fi
22:
23: DIR="$TMPDIR/`cpid`"
24:
25: while [ -d $DIR ]; do
```

```
26:   DIR="$DIR."
27: done
28:
29: mkdir -p $DIR
30:
31: INP="`basename $SRC`.pgm"
32: convert -geometry 128x128 $SRC $DIR/$INP
33:
34: cd $DIR
35: date >> detect_face3.log
36: detect_face3 $INP face.$INP >> detect_face3.log  2>&1
37: date >> detect_face3.log
38:
39: grep $INP GROUND_TRUTH | wc -l
40:
41: cd $_PWD
42:
43: rm -rf $DIR
44:
45: exit 0
```

A FDE can now call this server using the `serve` library calls:

```
 1: #include <connSERVE.h>
 2: #include <utilSERVE.h>
 3:
 4: float image_faces_detector(tokenpool *tp,cursor *c_location){
 5:   float res = FAILED;
 6:   char buf[BUFSIZ];
 7:   char *loc = NULL;
 8:
 9:   int FACEsocket = SERVEopen(
10:     get_prop_or("facehost","medusa.cwi.nl"),
11:     atoi(get_prop_or("faceport","4488")));
12:
13:   if (FACEsocket < 0) {
14:     fg_error("couldn't connect to FACE server\n");
15:     return FAILED;
16:   }
17:
```

```
18:   loc =  c_get_value(c_location);
19:   if (!strncmp("file:",loc,5))
20:     strcpy(buf,loc);
21:   else
22:     sprintf(buf,"file:%s",loc);
23:   strcat(buf,"\n");
24:
25:   fg_log("face::cmd=%s\n",buf);
26:
27:   SERVEputp(FACEsocket,buf);
28:   while(SERVEgetp(FACEsocket,buf,BUFSIZ)==0) {
29:     fg_log("face::reply=%s\n",buf);
30:     trim(buf,buf);
31:     if (buf[0]) {
32:       if (!strncmp("face:",buf,5)) {
33:         trim(buf,buf+5);
34:         tp_put(tp,"number",buf);
35:         res = SUCCESS;
36:         break;
37:       }
38:     }
39:   }
40:
41:   SERVEputp(FACEsocket,"quit");
42:
43:   SERVEclose(FACEsocket);
44:
45:   free(loc);
46:
47:   return res;
48: }
```

—

6.2 Importing ASCII dumps

Some detectors may produce a enourmous amount of tokens. For example a video feature grammar may contain an external detector which produces (several) histograms for each frame. For the external detector it may be cheaper to produce a semicolumn seperated dump of the feature data, instead of all `<partial path>:<value>` pairs as needed by the tokenpool. The `ascii2tp` tool helps

in these cases. The first line of the ASCII dump should contain format info, which the tools uses to produce the `<partial path>:<value>` pairs. This first line looks as follows:

`'#='` indicates that the line contains a format specification, the '#' should be the first character of the line.

`['#'<nr>'#']?<name>['?']?[';']?` the optional number indicates the times a token will be repeated, *i.e.* the collection size. If the number is negative, the first item in the dump contains the size of the collection. If the optional question mark is set the token is completely ignored. The semicolumn, which functions as a seperator, is only optional for the last token description.

To illustrate this cryptic header we will run the following ASCII file through the `ascii2tp` tool:

— *ascii.dump*

```
 1: #=id;size?;#-1#bin;#16#histo.bin
 2: 5;101376;9;396;1354;4789;54;0;0; ...
 3: ...
```

—

— *command line*

```
 1: (acoi 1) ascii2tp ascii.dump
 2: id:5
 3: bin:396
 4: bin:1354
 5: bin:4789
 6: bin:54
 7: bin:0
 8: bin:0
 9: bin:0
10: bin:0
11: bin:0
12: histo.bin:0
13: ...
14: histo.bin:0
```

—

This call can be embedded in a FDE using the same command as used by the `exec::` external detector: `FDEexec`:

—

```
1: float video_frames_detector(tokenpool *tp,cursor *c_filename) {
2:   char cmd[PATH_MAX];
3:   char *file = c_get_value(c_filename);
4:   sprintf(cmd,"ascii2tp -v %s",file);
5:   free(file);
6:   return FDEexec(cmd,tp);
7: }
```

—