YODA
User's Guide

for Version 0.9 preliminary

Florian Schwandt
Institut für Strahlen- und Kernphysik
der Universität Bonn

Updated for version 1.61
Olaf Diehl and Heiko Rohdjeß

February 5, 1998

Contents

Chapter 1

Introduction

This manual describes concepts, features and usage of the YODA program, an interpreter framework for online and offline data analysis. This section first deals with the motivation behind the YODA program and then gives you a short overview of YODA's features.

1.1 Why YODA ?

In general, data analysis for a small- to medium sized experiment in modern nuclear or particle physics can roughly be divided into two steps.

First, operations on *single (physical) events* are performed to transform raw data coming from the digitizing front-end electronics into physically meaningful data, possibly condense the information, do data reduction and so on. This means typically performing the same operations repeatedly on large amounts of sometimes heterogeneously structured data packets. This calculated and condensed information is then filled into statistical distributions, like histograms or x/y plots.

These *statistical distributions* are — in a second step — viewed on a graphics display, compared with pre-calculated distributions, integrated, projected, "fitted" with analytical functions, plotted and so on.

There are already some program packages available which satisfy the needs for the second step, more or less. First level data access, however, is normally done in an experiment dependent, hardcoded way. An executable program usually "knows" about the internal structure of the experimental data and does fixed calculation steps to produce histograms of primary data as well as more physical values (e.g. tracks, angles and momenta).

This traditional approach is sometimes necessary and natural, consider e.g. programs for event-displays or "number-crunching" algorithms. It also provides the fastest way of data processing in general. But, access to data which was not planned in advance (when the program was written), is usually cumbersome — if not impossible —, and almost always requires a local expert. To add or change functionality, the program has to be stopped, edited, recompiled, tested, probably changed again and so on until the desired changes are built in. For nontrivial programs, this may result in long turnaround times, making changes during "online" analysis impossible. Code for physical analysis sometimes "lives" close to code which implements the technical part of data access. As a consequence, sideeffects of user-written code may introduce subtle bugs into the analysis, which may become hard to track down. Finally, such a program is inherently non-portable to a different experiment with other requirements and is normally closely tight to the experimental setup.

On the other hand, first-level data analysis often needs the flexibility of a *programming language*. The calculations to be performed are highly dependent on the special experimental needs and may be of high (and increasing) complexity.

In general, arbitrary arithmetic operations must be performed, iterations on data (loops) must be possible, and logic decisions (if/else) must be made. In order to gain structure, the calculation should be broken into smaller units (subroutines and functions). Furthermore, calls for histogramming and plotting of both plain input data and calculated quantities must be available. "Cuts", which mean certain selection criteria on the data are typical tools for preparing physically meaningful information.

When dealing with physical analysis, the user should not need to worry about how to access the data. Instead, he or she should be provided with an data interface as simple and convenient as possible. If feasible, the user should be aided by mnemonic *names* for the data instead of abstract bunches of bits and bytes, and should be totally shielded from the inner details of event formats. This would also protect the data from being accidentally changed by the user.

A possible solution to the problem could be given by a dedicated, interpretative programming environment. It should be reduced to the physicist's experimental needs, while retaining the useful features of traditional programming languages. This programming environment should provide a well-defined and convenient interface to the experimental data. Analysis routines should be changeable "on the fly" to reflect the user's ideas instantaneously. Modifying sources should require no expert knowledge.

Unfortunately, this is not (yet) possible within the frame of an existing programming languages, but requires a "new', interpretative language with an interface to experimental data.

1.2 The YODA Interpreter

YODA is an abbreviation for **Y**et another **O**nline **D**ata **A**nalyzer, which is chosen in analogy to the name of the famous YACC program (**Y**et **A**nother **C**ompiler **C**ompiler).

YACC is a standard program in the UN*X programming environment, and generates compilers, interpreters and so-called *parsers* in general from a formal definition of the language to be used. YODA contains such an YACC-generated interpreter to give the users utmost flexibility in controlling their data analysis while retaining the comfortable features of an interactive program.

YODA enables the user to access experiment data coming from a tape, a disk file or directly from the running experiment, perform almost arbitrary processing on the data, and then create statistically accumulating objects like 1- or 2-dimensional histograms and define 1- or 2-dimensional cuts graphically during analysis.

Equipped with a programming language looking similar to `C`, and with functionality and simplicity at about the level of `BASIC`, users can realize their data processing in form of *user tasks*, which are (typically small) files containing the appropriate analysis procedures. Experimental data is accessed in form of simple *arrays* or scalar variables, with names freely chosen by the user to reflect their proper meaning in the experiment's context.

These analysis tasks can be loaded and unloaded dynamically during the running data processing. This allows the user to add new tasks without having to stop the analysis. It also provides the possibility for rapid development of analysis routines.

The generated histograms and plots can be displayed by the YODA program itself, or alternatively by a separate tools for visualization. This makes it possible to monitor the progress of analysis while it is running. This suits the needs for "online"-processing of data.

Apart from YODAs own display (named ZONI) PAW++ or pawX11 from the CERN program library ist used for histogram display. Data transfer between YODA and PAW can be accomplished via a shared memory area (not available for DIGITAL ALPHA architecture) or by files. See the discussion of PAW++ later in this manual. This is *not* a manual on PAW or PAW++, however.

The current version uses a data format as described in section 3, providing a comfortable means of structured data access. Especially rawdata coming from different subdetectors can be handled independently. Calculated or already pre-processed data e.g. from a simulation package can be integrated homogeneously and selected just as well as rawdata events. It is also possible to adapt the program to a different format of input data. People wishing to do so should read the appendix.

YODA's interpreter tries hard to weaken the major drawback of interpreters in general: the lack of speed in contrast to compiled programs. In fact, the user's analysis tasks are *compiled* into a pseudo-machine language, not interpreted line-by-line. This compilation process is normally invisible to the user (except for producing error messages for syntactically incorrect programs). For typical applications, a throughput of 100-1000 events per second on a cheap workstation was observed.

However, the YODA interpreter language has its limits. Track recognition in drift chambers, kinematic "fits" or high level trigger decisions involving a memory and/or cpu intensive treatment of the data is usually beyond the performance (and the abilities) of the interpreter language.

But for such applications, there exists a conventional programming interface for people who are familiar with `C++`, giving full and comfortable access to the incoming data as well as the histogramming features of yoda. Furthermore, the results of such a lengthy calculation can again be "fed back" into YODA, giving the user the chance to operate on them indistinguishable from true rawdata in the interpreter framework. For a closer look, see section 7.2

In this context, YODA can also be used to work as a *filter* program. Events can be analyzed and saved according to physical criteria for further analysis.

Sometimes it is sufficient to enhance the interpreter language by introducing new subroutines or functions, which are callable from the user's tasks and operate at full, compiled speed. Such an enhancement of the interpreter language is in fact very easily done (see section 7.1).

Even without such enhancements, the benefit of rapid changes to the analysis being possible often results in a significant gain in analysis speed, outweighing the speed-lack of interpreted code.

Chapter 2

Data Analysis with YODA

2.1 Overview

The simplified program structure is depicted in Figure 2.1. The YODA interpreter gets its data either directly from a file or from a data-serving program. This data-serving program can either be a formatter — for data not yet in the expected YODA data format — or e.g. a simulation program generating rawdata-looking events.

These two methods of input offers maximum flexibility. For the standard usage of YODA with input rawdata (typically written on a VME eventbuilder), the latter method is suitable, as it will be explained below.

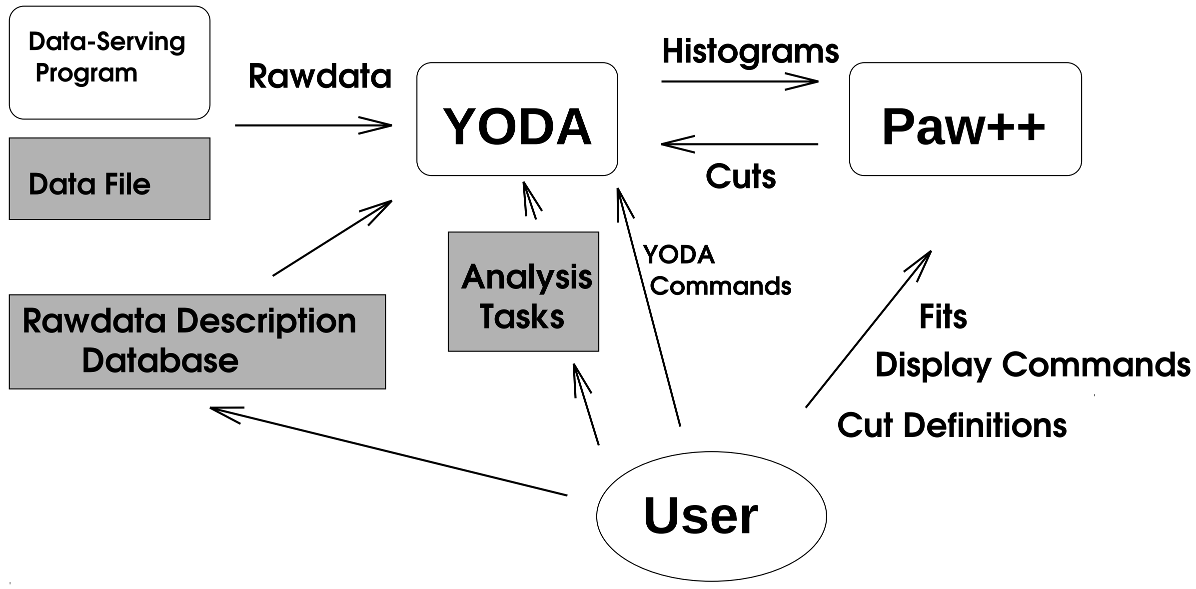

Figure 2.1: Simplified Structure of the YODA-Program and its Environment

Data processing is then done via analysis tasks written by the user for his special application. This is done in YODA's built-in interpretative language.

To provide convenient symbolic access to the data, the user has to create a database file, which describes the data coming in. Here, the user's knowledge about the experiment is put in.

Once YODA gets the data in one of the ways described above, it reads them event-by-event (each event normally corresponds to one hardware trigger and readout of front-end electronics). For each event, it looks up pieces of information on how to handle the event data in the user-supplied database. Then, it looks for user analysis tasks which act on these data, makes the data available to the analysis tasks, and calls the analysis task for this special event. Data for which no database information exists or no user task waits for, is simply discarded. Thus, the user can just "pick out" the data he or she needs.

Histograms are produced by the user task. These are periodically transmitted to paw and are available there almost indistinguishable from "normal" paw histograms. By now, histogrammed results are also saved using PAW's histogram I/O.

One can further use paw to define graphical cuts interactively. This will be explained in more detail in a later section.

As already mentioned in the introduction, there is also the possibility to enhance YODA via a programming interface (this is not shown in figure 2.1). When this approach is chosen, there is also the possibility to generate new events from inside YODA, or use YODA to a filter events, i.e. accept or reject events according to certain criteria and write them to file again. Details can be found in section 7.2.

2.2 Choosing the Input Mode

Rawdata is typically produced on a VME (Motorola 68K) front-end eventbuilder. Since the byteorder (the sequence of bytes inside a multi-byte integer or floating point value) given by the MC68K differs from byteorder an e.g. ULTRIX, integer rawdata must be "swapped", i.e. their byteorder must be reversed. This is an ugly technical problem and for the sake of must not be part of the analysis program.

To achieve this, a simple "front-end"-program called `server` is used. This program can be invoked by YODA at startup, and is passed as commandline arguments the requested mode of operation (disk, tape, or host). The server program tries to open the corresponding datasource, continuously reads from it, "swaps" the data into the proper byteorder and passes the data to YODA via the stdout/pipe mechanism.

This interface of reading another program's stdout is also suitable for using YODA with a different type of rawdata format. With simple calls to library routines provided by YODA, any data format can be made YODA-readable.

If there is data to be processed which was either produced by a UN*X program in the first place (e.g. a simulation program), or by YODA itself (using YODA as a data filter, see section 7.2), the file can be opened directly.

2.3 Information Supplied by the User

To be an experiment-independent framework, two pieces of information must be supplied to YODA by the user at runtime:

- What does the data look like ? (e.g. how many ADC values are expected together with a certain hardware trigger ?)
- What kind of analysis should be performed ? (e.g. what are the interesting quantities to be filled into histograms ?)

The answer to the first question is read from a plain text file called the *Selector Database*. For every piece of data which is to be investigated, there must be an entry which gives the data a name, declares its datatype (e.g. floating point or integer), specifies the "tag", which identifies the data in the incoming datastream, and defines the maximum size of the expected values.

This file has to be created new or at least modified for each specific experimental setup. To create one from scratch, you need to know a little about YODA's event format, so detailed information is postponed here.

The answer to the second question is given by the *user tasks*, which contain the analysis steps written in the YODA interpreter language, later referred to as the *hoc language*. Once the selector database is set up for a given experiment, many different user tasks can operate on the data, investigating different aspects of the experiment. These tasks can be loaded, unloaded, edited, en- or disabled freely during the running analysis. More than one task, probably operating on different types of events can run independently at the same time.

Each user task contains code for initialization, code to be processed whenever an appropriate event comes in, and optionally code which is executed when YODA is stopped or all events have been processed.

These files are created or modified during the analysis process. See chapter 5 for a complete description of the interpreter language. Some examples for "hoc-files" can also be found in section 5.

2.4 Results, Histograms and Cuts

Aided by the above information, YODA does (in fact, the user routines do) the required data processing, and produce histograms or calculated values as results. Calculated values can simply be printed out, appearing on the controlling terminal. To do formatting, the full `printf` functionality can be used.

Histograms are just filled by YODA, and can be inspected by the internal display or may be copied to a shared memory area between YODA and the displaying process (PAW++; currently not supported on at least DEC UNIX). The calls for defining and filling histograms are very much like the calls from the CERN hbook library (apart from the transfer to paw, no hbook routines are actually used). In most cases, however, YODAs own display will be sufficient for data visualization.

Cuts (one- or two-dimensional criteria on the data, often also known as *gates*), are also declared by the user tasks. Their actual definition, however, is read from files. PAW can be used to produce the definitions interactively and save them to file. These definitions can be read from the user task or interactively via YODA's pulldown-menu.

This also allows the user to change them "by hand" or with another text processing tool, since the cut definitions are plain ASCII format.

Chapter 3

Data Format

Since the data format used by YODA is the same as the TDAS ISKP data format, the information here is mainly taken from [sch94]. Consult this manual for details on the TDAS data acquisition software currently in use at the ISKP.

When using YODA, you normally need not worry about the "low-level" binary structure described here. You should, however, have a basic understanding of *containerevents* and *subevents*.

3.1 Containerevents and Subevents

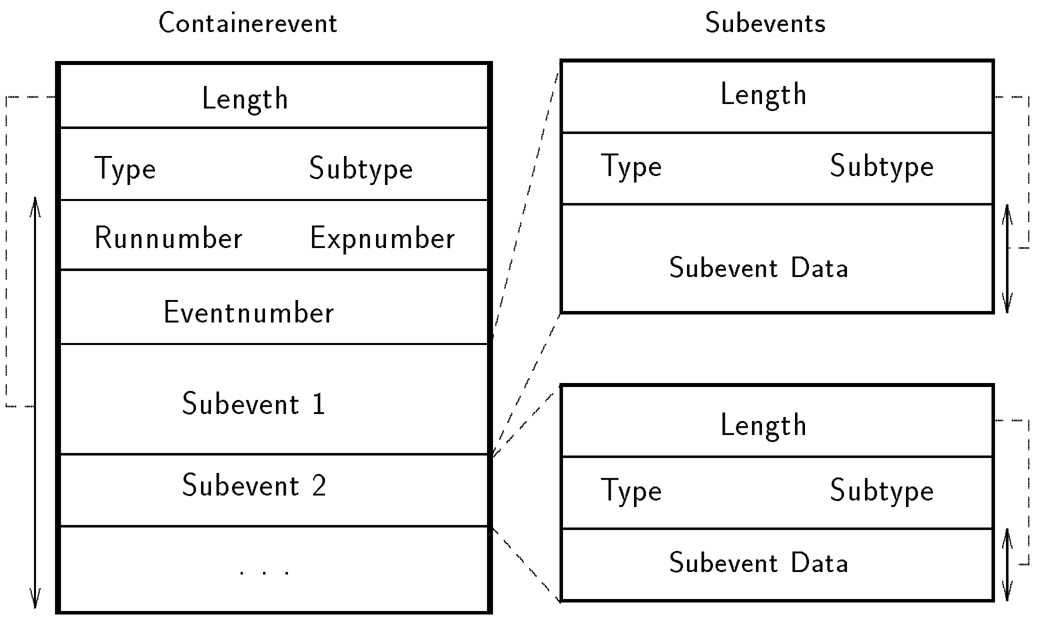

Figure 3.1: Layout of Container- and Subevent structure. The width of a box is 4 bytes

The binary structure used by YODA for representing physical events is shown in 3.1. There are two main classes of eventstructures: *containerevents* and *subevents*. Both share the same header information: a 4 byte unsigned integer giving the total length (in bytes) of this containerevent / subevent (*excluding* this header), and a type/subtype identification (each an unsigned 2 byte-value), which is used to classify this event. This header is then followed by data, which meaning is given by the type/subtype information field. The important difference between container- and subevents is the fact that containerevents contain data + subevents, while subevents carry only their data.

Not all defined types/subtypes are used, but the user can (in principle) use all defined types or add new types/subtypes according to his or her needs. Both types and subtypes are actually 16-bit values (shorts), but are given in a symbolic notation here for readability.

Type and subtype information is used to label the data according to its meaning. In YODA's selector database, this information is needed to identify the data and give it its name, under which it it accessible inside the user tasks.

3.2 Accessing Data from YODA

As it was said in the introduction, the user should not be concerned with technical aspects of data processing. This is — to a large extent – true for the current version of YODA. To be independent of the experimental setup, the user must, however, provide some "technical" information once for each experiment. This will be discussed below.

To access data inside the subevents, YODA uses a user-supplied ASCII file which connects data fields with symbolic names. This file is called the *selector database*, and must exist in your working directory.

It contains a one-line entry for each distinct data element. Each line has the following form (lines beginning with `#` are comment lines and are ignored):

```
NAME DATATYPE CONTAINERTYPE C.SUBTYPE SUBEVENTTYPE S.SUBTYPE BYTEOFFSET [DIM]
```

Each such line is called a *selector*. (The square brackets around `DIM` denote an optional value.)

`NAME` is a symbolic name chosen by the user, which makes the data accessible to the user tasks. The data described by the selector is copied to a variable with the name `NAME` into the set of variables for every user task, which has "linked" to this selector. Then, the user task is called and can process the data. After that, the next containerevent is processed. If another match occurs, the variable is overwritten with the new value, the user task is called again and so on.

Once a user task chooses to investigate a certain piece of the data (defined by a single line entry in the selector database), the incoming containerevents are scanned by the YODA program to match the indicated combination of containerevent type and subtype and subevent type and subtype. If one of these fields is not significant for the selection, it can be chosen to be ignored by specifying a `0` (zero) for the corresponding type or subtype field.

To select e.g. each and every subevent from the incoming datastream, you would specify a combination of four zeroes for the containerevent- and subevent- type and -subtype fields. (You would then — except in rare cases — have problems determining `DATATYPE` as well as `BYTEOFFSET` and `DIM`, see below).

When a subevent with its enclosing containerevent matches the type and subtype criteria, the data is regarded as `DATATYPE`, where `DATATYPE` can be one of `char`, `uchar`, `short`, `ushort`, `long`, `ulong`, `float` or `double`. This set corresponds to the standard datatypes of the "C" language, with `uchar`, `ushort` and `ulong` meaning `unsigned char`, `short` and `long`, respectively.

The last but one field in the line specifies the offset in bytes from the beginning of the selected subevent's data, where the desired data is to be found. This provides a means for skipping unwanted data inside a subevent, or addressing heterogeneously structured data.

The last field (`DIM`) determines the number of elements (all of type `DATATYPE`, see above), which is to be selected from the subevent's data field. This implies, that `NAME` is to be used as an array (a vector) inside the user tasks. This feature is used to address homogeneously structured data, e.g. a set of ADC values belonging to subsequent channels, which will all be processed in the same way.

If the last field is (DIM) is omitted, NAME is used as a "simple" variable inside the user tasks.

Your front-end readout routine (or your simulation package, whoever produced the data) determines which types and subtypes actually occur in YODA's input datastream. Look at e.g. your readout routine to see which types and subtypes are employed to label the containerevents and build your selector database accordingly. (Probably, this is already done for you. Ask your local expert).

If you have no idea which types/subtypes are actually used, you can use an event dumping program like e.g. the `server` program. This program is used for the setup at the ISKP to swap the rawdata into the correct byteorder. It can also be used to dump the events as they are processed.

If the data is e.g. coming from a tape devnrmt0h/ in tar format, you would enter:

```
server -t /dev/nrmt0h -v | more
```

This will provide you with a full dump of the rawdata events, listing all type and subtype information.

If you have no idea what the data *means*, you're in trouble anyway. You then have almost no chance to do suitable analysis with it. (This situation is about to be improved. Some efforts are going on to make the data format somehow "self-documenting", so the user does not even know about types and subtypes. Instead, they are chosen automatically by the front-end data producing program (e.g. the data acquisition) and hidden inside special "info"-containerevent.)

3.3 YODA as a Filter Program

As mentioned before, YODA can also be used to write data. The data format is the same as discussed above, so naturally files written by YODA can also be read by YODA.

Because the main purpose of YODA is to analyze the data and create histograms and results, this is not the default behavior. If you need such a functionality, you must create your own, dedicated version of YODA with some tiny modifications. See section 7.3 for details.

Chapter 4

First Steps with YODA

This chapter should give you a gentle introduction into data analysis with YODA. It's a sort of "mini-tutorial", not a reference. Its best usage is in front of a workstation running both YODA and PAW++. Things are kept very simple to convey the basics. After the principle operation is understood, the reader should then read the section on YODA's full command set and language, probably diving right into his or her "real" physical data. Users who are already familiar with YODA's basics should probably skip this section.

4.1 Preliminaries

First, you should check whether YODA is installed on you system. For high level graphics you will also need PAW++ (in principle, everything can be performed using the plain vanilla version of paw, but PAW++ is highly recommended). Enter `which paw++` and `which yoda` to see if these programs are found in your `PATH`. If not, contact your local expert.

YODA uses the environment variable YODAHOCPATH to search for your analysis tasks written in the `hoc` language. If it is unset YODA will first look in your current working directory. Assuming your hoc-files reside in `/usr/users/joe/hoc` you should type:

- for csh and tcsh users: `setenv YODAHOCPATH /usr/users/joe/hoc`
- for sh, ksh, bash users: `YODAHOCPATH=/usr/users/joe/hoc;   export YODAHOCPATH;`

(If you do not know your shell, enter `ps`, which should give you (among other things) the name of you shell). You should do this in your `.login`, `.profile`, or whatever the appropriate startupfile for your shell is. If you're in doubt, contact your local administrator.

4.2 A Sample Selector Database

To follow the example below, you must further have a selector database file with an entry for our data in the example. There is a default `selector.db` file installed with any YODA version. To find out where it resides type: `yoda-v` this will print out somthing like

```
YODA Version 1.5 from May  2 1997 16:47:31
Search Path is .:/usr/local/yoda
```

The last item in the search path is where you should look for a `selector.db` file. You can copy it into your current working directory and modify it to fit your needs. It should contain the `DUMMY` line listed below for the example to work (the lines starting with `#` are comments and are ignored by YODA.:

```
#
# selector.db: YODA's selector database file
#
# NAME DATATYPE CONTAINERTYPE C.SUBTYPE SUBEVENTTYPE S.SUBTYPE BYTEOFFSET [DIM]
#
DUMMY   ulong   3 0 2050 3 0 16
#
# each line not starting with a '#' defines a SELECTOR, which provides a means
# for a defined data access to the TDAS containerevent/subevent structure from
# the YODA interpreter language. each SELECTOR, if found in the eventstream,
# becomes the name of a scalar variable (or array, see DIM below), which
# contains the actual data.
#
# NAME:          a unique string to identify the data. this will also be the
#                name for the YODA variable, which will contain the data.
# DATATYPE:      can be one of  { char , uchar, short, ushort, long, ulong,
#                float, double }. this field determines the datatype as it is
#                to be found in the events
# CONTAINERTYPE:the containerevent-typefield value, which must be matched
#                to select the current containerevent for processing
# C.SUBTYPE:     the containerevent-subtypefield value, which must be matched
#                to select the current containerevent for processing
# SUBEVENTTYPE: the subevent-typefield value, which must be matched
#                to select the current subevent for processing
# S.SUBTYPE:     the subevent-subtypefield value, which must be matched
#                to select the current subevent for processing
# BYTEOFFSET:    the offset in bytes from the beginning of the subevent's
#                data area, where the data is found. this makes it possible
#                to skip arbitrary amounts of data as well as to define
#                selectors for one subevent, each with a different
#                BYTEOFFSET for addressing heterogeneous datastructures
# DIM:           the dimension of the selector, which means the number of
#                dataitems of type DATATYPE which is to be put into the selector
#                variable. DIM > 0 means selector is an array of type DATATYPE
#                with (at most) DIM entries. for variable-length subevents,
#                the actual length is supplied in the scalar variable
#                NAME_LEN, with NAME as described above.
#                this field is optional. if left out, a scaler variable
#                is assumed
```

This database defines `DUMMY` as pieces of data, all of type `unsigned long` (32-bit integer values without sign), identified by the 4-tuple `3 0 2050 3`. Refer to the section 3.2 for details. For the moment, just take this information as a given "tag" to identify the data in the incoming data stream. The next number (0) means that the dummy-data starts with no offset to its "envelope" (i.e. no data is to be skipped). The last number (16) means that DUMMY is a field of (at most) 16 subsequent values (each of type `unsigned int`).

As this line is sufficient to describe our (dummy) data, we now need a file which contains our analysis task.

4.3 A Sample Analysis Task

Suppose, we want to to calculate the mean value and of our 16 imaginary data values coming in, and print them out to the screen. If we denote the values belonging to a *single event* with `DUMMY[0]`to `DUMMY[15]`, the following user task calculates the above quantity for each event (create it with an editor - remember the ; at the end of each statement, and save it under the filename `dummy.hoc`):

```
/* procedure init. called once when this task is loaded */
proc init()
{
select DUMMY;     /* look only for data of signature DUMMY */
}

/* procedure analyze. called every time a DUMMY event is coming in */
proc analyze()
{
  for(i = sum = 0; i < 16; i++)        /* loop over all array elements */
    sum += DUMMY[i];
  mean = sum / 16;                     /* calculate the mean value */
  printf("mean value is: %f\n", mean); /* print out the mean value */
}
```

Let's take a closer look at this file. At first glance, it looks like `C`, but some things are special. There is no `main()` function, instead there are two subroutines here denoted `init()` and `analyze()`. For each task, the convention is that the procedure `init()` is run once whenever the task is loaded into the YODA interpreter, whereas the procedure `analyze()` is run each time a matching event occurs in the incoming datastream. There is also an optional procedure `finish()`, which is called each time the data processing is halted (either manually, or at the end of the input file).

How does YODA know when to call the procedure `analyze()` ? The `init()` procedure contains the statement `select DUMMY;`, which does two things:

- It looks for the `DUMMY` line in the selector database (see previous section), and (if found) remembers that this task is to be called whenever the `DUMMY` data is found in the incoming datastream.
- Each time the data is found in the input datastream, it makes it accessible to the procedure `analyze()` in form of an *array* with the same name as the selector (i.e. `DUMMY`. Addressing the elements of an array is done just as in `C`, that means with square brackets (`[]`).

So the effect of this `select` statement is, that each time DUMMY data are found, `analyze()` is called, and at the same time the `DUMMY[]`-array contains the actual values. For the next incoming `DUMMY`-data, the array values are updated, `analyze()` is called again, and so on.

The rest of the file is not hard to understand. Note that all variables (and arrays) in the analysis task are of type `double`, no matter what the type of the *input* data (as described in the selector database) is. So the format specification in the `printf`-statement is of floating-point type, This must even be used for "integer-looking" variables as `i` or `j`.

Now we are ready to try this dummy-analysis.

4.4 A Sample YODA Session

First, change to the directory where the file dummy.hoc resides. To run our simple user task, start YODA from the commandline with:

`yoda`

You should see something like the following:

The (xterm) window, where you typed the yoda startup command becomes the *terminal window*, providing a commandline-oriented input/output facility to YODA. You can do all

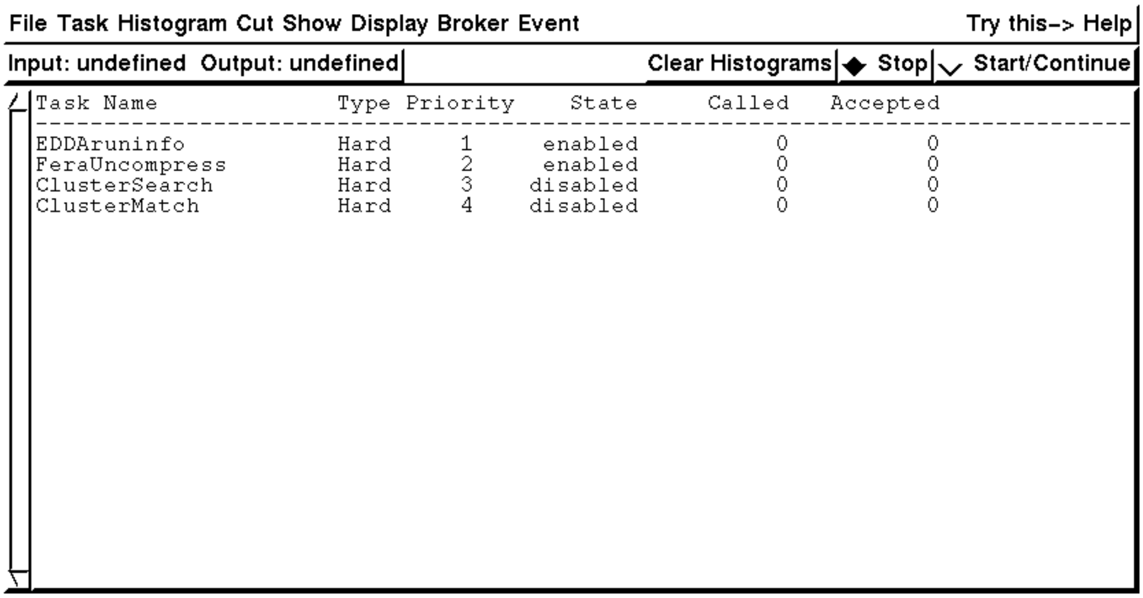

Figure 4.1: Starting YODA

available YODA actions from here. Alternatively, you can use YODA's *main window* (see figure 4.1), which provides a somewhat more comfortable graphical interface to YODA's functions. It consists of some *pull-down menus*, some *push- or radiobuttons*, and a rather large status window, which is mainly empty at starup. The latter is updated periodically to provide information on runs, tasks, histograms and cuts, as requested. You can select the information to be displayed with the `Show` pull-down menu.

Now, select the entry `Open Inputprocess` from the `File` pulldown-menu. In the dialog box showing up, you enter the following input command:

```
server -D 20
```

This starts the data-serving program. The options given activate the dummy-data mode, which is made only for testing purposes. YODA then generates all the data itself (just dummy-events containing random values, but with the data description as in this example). The number means the number of events generated per second.

If this screen (or a similar one) doesn't show up, you're in trouble. Check the following:

- If the error is something like: `yoda: command not found`, check your `$PATH` environment variable. It must contain the path of the YODA executable.
- If you obtain a message like `invalid command name "wm"` your DISPLAY environment variable is not correctly set and YODA is unable to open a window. Ask your local expert for details.

To run your analysis task, it must first be *loaded* into YODA. Do so by choosing the entry `Load...` from the `Task` pulldown menu. A fileselectbox opens and offers you the files in the current directory to choose. Click on `dummy.hoc`, and click `OK`. YODA now reads the file `dummy.hoc`, checks it for syntax and — if successful – adds the task to its list of active tasks.

If you see error messages in the terminal window (most likely typing mistakes or missing semicolons), the task will not be loaded. Fix the errors (unfortunately, the error description is not very verbose. Usually, at least the line number should be helpful), and try again.

Eventually, the task should appear in YODA's status subwindow with its name and with the indication "active".

Now, click on the `Start/Continue` radiobutton located at the right of YODA's main window. You should see the task being called (watch the `called` column in the status display, and the `accepted` field increasing as well. This means that events are being dispatched to your analysis task.

Since you put the `printf(...)` into the procedure `analyze()`, you should also see the desired output in the terminal window. Don't worry about obviously random values appearing, since the input (`DUMMY`) data is in fact generated randomly.

Now you can play around with the various YODA functions. Try to en- or disable the task, try to reload it a second time, choose different information to be shown in the status window (use the `Show` pulldown menu), and try to stop the data processing by clicking on the `Stop` radiobutton. You can resume the analysis by clicking `Start/Continue` again. If you have enough, you can leave YODA by choosing `Quit` from the `Task` pulldown menu.

Now you're ready to produce some histograms.

4.5 Producing Histograms

Histograms are statistical objects counting the frequencies of a independent variable or tuples in a set of intervals.

Histograms are realized in YODA as one- or two-dimensional objects, either counting the frequency distribution of a em single variable or of a 2-tuple of values, respectively.

To see how the histogramming options work, create a second analysis file called `dummy2.hoc` with a demonstration of most common usage of the histogramming tools. It should contain the following:

```
/* procedure init. called once when this task is loaded */
proc init()
{
select DUMMY;
h1 = histo ("SingleSpectrum", 1024,  0, 1.);
h2 = histo2d("TwoDimSpectrum",  64, -1, 1, 64, -2, 2);
}

/* procedure analyze. called every time a DUMMY event is coming in */
proc analyze()
{
hfill  (h1, sin(DUMMY[0]));
hfill2d(h2, sin(DUMMY[1]), 2.*cos(DUMMY[2]));
}
```

This tasks shows an example of creating and filling histograms. In the procedure `init()`, two histograms are defined: a one dimensional, created with the call `histo(...)`, and a two-dimensional with the call `histo2d(...)`. Both calls return a *histogram handle*, which is later used to refer to the created histogram. These handles are saved in the user variables `h1` and `h2`. Note that these handles have nothing to do with PAW's *histogram identifiers*, but are internal YODA parameters. After creation, all channels of the histograms are zero.

The parameters to `histo(...)` are the following: The first one is a string (in double quotes), giving the *name* of the histogram. This name will automatically be preceded with the name of the task in which context the histogram is created and a period ("."). So in this case, the full name of the first histogram is `dummy.SingleSpectrum`. The next parameter (1024) defines the number of *channels* (bins) the histogram should have. The next and the last parameter (0, 1) define the lowest and highest x value which is expected to be filled into the histogram. The histograms's x-axis then runs from 0 (low edge of first bin) up to 1 (high edge of last bin). Note, that in this way the x-axis can be of arbitrary floating point values, and need not be integer channel numbers. So this histogram definition divides the interval between 0 and 1 into 1024 equally spaced bins.

The parameters to `histo2d(...)` are similar: First comes the name of the 2-dimensional histogram, then the number of channels in x (64), low and high limits in x (-1, 1), number of channels in y (64), and low and high limits in y (-2, 2). The corresponding (square) area between (-1, -2) and (1,2) is divided into 64x64 channels (bins).

The procedure `analyze()` shows the typical usage of the so defined quantities: values are "filled" into the histograms via the calls `hfill(...)` or `hfill2d(...)`, respectively.

The first parameter is the *handle* of the histogram as it is returned by the appropriate call to `histo(...)` or `histo2d(...)`.

For the one-dimensional case, the second (and last) parameter is the value which is to be filled in. Here, the sine of the first (random) dummy-array element is investigated. If the actual value is in the interval between -1 and -1 + 1*(2./64)), it will be filled into the first channel of the histogram, causing the counter for bin 0 to be incremented by one. If it is in the interval between -1.+ 1*(2./64) and -1.+ 2*(2./64), the second channel is incremented and so on.

For the two-dimensional case, the second and third parameter is the (x,y) tuple which is to be filled in.

Now let's see how it works!

4.6 Viewing Histograms

If YODA is not active, start it first as you did before. To shorten things up, you can also select the input datasource with appropriate commandline parameters. Type:

`yoda -p "server -D 20"`

This should give you the same result as starting YODA first and then selecting the input-process to be `server -D 20`.

Then, load the example above into YODA, and push `Start/Continue` state. You should see that events are being processed. Furthermore, if you select `Histograms` from the `Show` pulldown-menu, you should also see the two histograms that you defined being displayed. Note the increasing number of entries as the analysis goes on. Now it's time to look at them graphically.

The easiest way to do this job is to use the internal display of yoda. At first, you have to enable it by choosing *on* from the *display* menu. Now you have two more windows: one titled "zoni" and the other titled "display". The zoni is a very simple histogram manager. It is used for tasks concerning the selection and arrangement of histograms. The display is a "canvas" showing the selected histograms. Also, it allows you to act on one instance of a histogram, e.g. performing a zoom.

Now you can browse your histograms by clicking with mouse button three on the desired histogram in the zoni listbox. If you want to view more than one histogram at once, use the button matrix to enable displaying that number of histograms. Histograms are added by double-clicking with mouse button one, while button three replaces the current one. Further capabilities like zooming, logarithmic axes, probing, printing etc. are described by the yoda help menu.

Another possibility is to use PAW++ for visualization. The first way is to write an hbook file and read this file with PAW++. Histograms are written using the Histogram menu or by typing

```
histowrite filename
```

in the xterm yoda is running in. This is fine if you are interested to save your histograms for future use. The other way is to use histogram transfer via global section (not available on all platforms).

To use the global section, it must be enabled when starting up yoda, e.g.

```
yoda -t 15 -p "server -D 20"
```

which will update the global section every 15 seconds. Then start PAW++ by typing:

```
paw++
```

Then, three windows should appear additionally to your existing YODA and terminal windows.

In the *Input Pad* of the *Paw++ Executive Window*, enter:

```
global yoda
cd //yoda; hrin *
```

These commands tell PAW++ to link itself to the shared memory area created by YODA (which is addressed by its name: yoda), then to change to this shared memory as the current working space, and then read in all histograms. The histograms should then appear in the `PAWC` area. Check this by clicking on the `PAWC` entry in the *Paw++ main browser*, and see histograms appearing in the space beneath it. Then, double-click on one of the histograms to get it displayed in the graphic window. You can display the other one as well by double-clicking on its entry in the main browser.

To update the state of the histograms (which should still be being filled in the meantime), recall the command

```
cd //yoda; hrin *
```

with the cursor keys in the executive window (or reenter the command), to get the current contents of the histograms into the `PAWC` area.

You can now redisplay, fit, copy, add, print or whatever you want to do with them. See a PAW++ manual (if you can get one) for details.

To get a continuously updating display, you should define yourself a `loop` macro in a file e.g. like this (in a file loop.kumac):

```
oncemore:
zone 1 2
cd //yoda; hrin *
cd //pawc
h/plo *
wait 5 ' '
h/del *
goto oncemore
```

If this is executed by entering `exec loop` or clicking on the appropriate entry from the `macro` section in the main browser, the histograms will be shown updating in a 5 second interval while they are being filled.

You can stop this any time by typing `^C` (Control-C) in the executive window.

While this should be sufficient for a first try, you should now either play around on this level (using YODA's dummy mode), or get right into your "real analysis".

You should read the reference section next to get an overview over YODA's complete command set and features.

Chapter 5

The Hoc Language

This section should serve as a *reference manual* concerning the *hoc language*, which is used for the analysis tasks.

Hoc means **h**igh **o**rder **c**alculator, and is a modified version of Kernighan's interpretative arithmetic language, as described in [ker84]. It has been modified in several respects to make it suitable for physical data processing. The similarity to the `C` programming language has been enhanced, new operators have been introduced, and special built-in functions for histogramming, "cutting", and data selection have been added.

Eventually, hoc will become "as close to `C` as possible, but no closer" [str93]. This reference manual here describes the proposed version 1.0. Versions prior to 1.0 may lack some language elements, but this will be indicated and should only have minor impact on the language abilities.

All arithmetic is performed using *double precision*, i.e. 8-byte IEEE floating point values. This is also the only data type available inside hoc. *There are no integers, logicals or character variables available !* There are, however, integer *looking* constants as well as string constants enclosed in double quotes.

5.1 Arithmetic Expressions

Hoc as described here is based on arithmetic expressions, similar to `C`. The grammar for these expressions is the following:

```
expr:   number
    |   variable
    |   variable = expr
    |   array [ expr ]
    |   ( expr )
    |   expr binop expr
    |   unop expr
    |   function ( arguments )
```

(This is to be read: " An `expression` is formed by *either* a `number` *or* a `variable` *or* an array-name followed by `[` followed by an `expression` followed by `]` *or* ...)

Note that this notation is recursive in nature, allowing familiar expressions like e.g. `sin(exp(b[i] * -1.5E-3))`.

Numbers are either *floating point values* in their usual notation (see *scanf(3)*): digits, decimal point, digits, `e` or `E`, exponent with sign, or written in hexadecimal notation just as in "C" (e.g. `0xFF` meaning `255`). Note, however, that hex constants are stored as floating point values as well.

All *identifiers* (i.e. names for variables, arrays or functions) consist of a letter, followed by an arbitrary number of letters, digits or "_" (Well, the maximum length is 128 characters).

`binop` means a binary operator (involving two operands), like addition or test for equality, `unop` means a unary operator like `!` (logical NOT) or unary minus (switching the sign). See 5.1 for a list of available operators.

<table>
<tr><th>Oparator</th><th>Purpose</th></tr>
<tr><td>^</td><td>Exponentiation (like FORTRAN **</td></tr>
<tr><td>! -</td><td>(unary) logical and arithmetic complement</td></tr>
<tr><td>* / %</td><td>multiplication, division and modulo</td></tr>
<tr><td>+ -</td><td>addition and subtraction</td></tr>
<tr><td>>> <<</td><td>bitwise shift left, shift right</td></tr>
<tr><td>> >=</td><td>comparison: greater than, greater or equal</td></tr>
<tr><td>> >=</td><td>comparison: less than, less or equal</td></tr>
<tr><td>== !=</td><td>comparison: equal, not equal. (all the same precedence)</td></tr>
<tr><td>&</td><td>bitwise AND</td></tr>
<tr><td>|</td><td>bitwise OR</td></tr>
<tr><td>&&</td><td>logical AND</td></tr>
<tr><td>||</td><td>logical OR</td></tr>
<tr><td>++ --</td><td>(post- and pre-) increment, decrement</td></tr>
<tr><td>= += -= *= /=</td><td>assignment</td></tr>
</table>

Table 5.1: List of hoc Operators with Decreasing Priority

Logical expressions have the values 1.0 (true) and 0.0 (false). Like in `C`, every value not equal to 0.0 is considered true. Note, that due to the floating point representation, every result coming from a computation cannot be guaranteed to be exactly zero! There are also some predefined constants available, see 5.1.

5.2 Statements and Control Structures

The following grammar shows the basic building blocks of hoc programs

```
stmt:   expr ;
     |  select NAME;
     |  access NAME;
     |  generate NAME;
     |  set_priority NUMBER;
     |  dim NAME indexlist ;
     |  variable = expr ;
     |  procedure ( arglist ) ;
     |  for( expr ; expr ; expr ) ;
     |  while ( expr ) stmt ;
     |  break;
     |  continue;
     |  if ( expr ) stmt ;
     |  if ( expr ) stmt; else stmt ;
     |  return optional-expr ;
     |  { stmtlist }

stmtlist: /* empty */
     | stmtlist stmt

indexlist: /* empty */
```

Function	Purpose
`sin(x)`	sine of x (in radian)
`cos(x)`	cosine of x (in radian)
`tan(x)`	tangent of x (in radian)
`asin(x)`	arcus sine of x
`acos(x)`	arcus cosine of x
`atan(x)`	arcus tangent
`atan2(y, x)`	arcus tangent of y/x
`sinh(x)`	hyperbolic sine
`cosh(x)`	hyperbolic cosine
`tanh(x)`	hyperbolic tangent
`log(x)`	natural logarithm of x (base e)
`log10(x)`	logarithm of x base 10
`exp(x)`	exponential of x (e^x)
`sqrt(x)`	square root of x ($\sqrt{x}$)
`int(x)`	truncates floating point part of x
`abs(x)`	absolute value of x
`ceil(x)`	smallest integer not smaller than x
`floor(x)`	largest integer not greater than x
`fmod(x,y)`	computes the modulo remainder
`randomFlat()`	random number, flat distributed in (0, 1)
`randomGauss()`	random number, gauss distributed

Table 5.2: List of Available Built-in Arithmetical hoc Functions

```
     | indexlist index

index: [ expr ]
```

Syntax and semantic of these control-structures are just like in `C`.

The `select` statement has the special meaning discussed in the previous chapter. It tells the YODA when to call this task, and prepares the availability of the online data via the variable (or array, depending on the selector database) NAME.

The `access` statement is similar to the `select` statement in that it makes data available as an array. However, it does *not* decide wether this task is called or not.

Quite special is the `generate` statement. Here a subevent (specified by its NAME in the selector database) is declared to be generated by the callback. An array NAME and a variable NAME_LENGTH holding the dimension given in the database are automatically created. The user then can change the contents of the array and the length of the subevent which is actually written to the eventstream after the analyze procedure has been called.

Closely related to this feature is the command `set_priority` which allows to change the default priority of 50 to any value in the range 1-99. A lower number means that the callback is called earlier. This allows to create subevents by one callback whcih can then be selected or accessed by callbacks (soft or hard) of higher priority. The priority of a soft callback cannot be changed via a task reload. The task must be unloaded and then loaded again in order to make the new priority active.

Variables need not to be declared. All variables are of type `double` and are global (i.e. every function or procedure of the same task can access the variable for reading and writing).

To define simple arrays (of type `double`, as all variables are), use the `dim` statement. It defines the following NAME to be an array with the dimension `expr`. Note that arrays are indexed just as in `C`, starting with zero with the last valid index is $dimension - 1$. Note also that whole arrays cannot be passed to procedures or functions.

`printf(format,...)`
`sprintf(format,...)`
 	
`read(varname)`
 	
`xread("description",default_value)`
 	
`xread_string("description","default")`
 	
`read_string("description")`
	print arguments to stdout according to `format`
	print arguments according to `format` into a string
	the string is returned (only allowed as argument to other functions)
	read a number from the terminal window and assign the value to variable `varname`
	read a number from an X window (which will pop up). Returns entered value, the description string is mandatory, the default value may be ommitted.
	read a string from an X window (which will pop up). Returns pointer to entered string, the description string is mandatory, the default value may be ommitted.
	read a string from the terminal. Returns pointer to entered string, the description string is not mandatory.

`fp=fopen("filename","mode")`
 	
`fclose(fp)`
`fprintf(fp,"format",...)`
	open a file with name `filename` for read (mode=r), write (mode=w) or append (mode=a) and return file pointer
	close a file opened with `fopen`
	like `printf`, but print to file and not to stdout.

`id = ijopen("filename")`
`val= ijget(id, x,y)`
`ijclose(id)`
	opens a IJ_data lookup table file
	make a lookup at (x,y)
	close a lookup table file

`exit_yoda()`	terminates yoda

`exp_num()`
`run_num()`
`evt_num()`
`evt_size()`
`write_event(level)`
 	
`abort_event()`
	the current experiment number
	the current run number
	the current event number
	the size of the current event in bytes
	write the current event to the YODA output stream. If `level` is given subevents created by callbacks with priority value greater than `level` are not written. Default `level` is 0 (only raw data)
	abort processing of the current event, subsequent callbacks are not called

`dedx(energy)`
 	
`setup_dedx(I,Z/A,Rho,z,M,a,m,X1,X0,-C)`
	calculates the differential energy loss for plastic scintillator (in MeV/cm) for protons of kinetic `energy` (in MeV/c^2).
	setup parameters for subsequent calls to dedx, where
	I = mean excitation potential (eV)
	Z/A = atomic number of absorbing material/ atomic weight of absorbing material
	Rho = density of absorbing Material (g/cm^3)
	z = charge of incident particle in units of e
	M = mass of incident particle (MeV) parameters for density correction: (see leo's book for examples) a, m, X1, X0, -C

`shell("sh shell command")`	execute string as shell command, using `sh`

Table 5.3: List of Available Miscellaneous Functions

Name	Value	Meaning
PI	3.14159265358979323846	π, ratio of circle circumference to its diameter
E	2.71828182845904523536	e, base of natural logarithms
GAMMA	0.57721566490153286060	Euler-Mascheroni constant
DEG	57.29577951308232087680	$180/\pi$, degrees per radian
PHI	1.61803398874989484820	$(1+\sqrt{5})/2$, the golden ratio
PID	arbitrary	current pid of YODA

Table 5.4: List of Available Built-in Constants

YODA also supports multidimensional arrays. They are declared like `dim x[10][5]`.

5.3 Procedures and Functions

Procedures and functions are distinct objects, although they are defined in about the same way. The difference is used to perform error-checking at runtime: a procedure must not return a value, but a function has to.

The syntax for function and procedure definitions is:

```
function:  func NAME() stmt

procedure: proc NAME() stmt
```

In this syntax, `func` and `proc` are keyword and must be entered literally. `NAME` stand for the name of the function or procedure, respectively. `stmt` is meant in the sense of the previous section.

When calling functions and procedures, arguments separated by commas can be passed in brackets. These are referenced inside the function or procedure just like in e.g. the bourne shell: `$3` is used to reference the third argument. Argument counting starts with 1.

Arguments are passed as values (so arbitrary arithmetic expressions are possible) and are inside the function or procedure semantically equivalent to variables.

To reference an argument with number greater than the number of arguments actually passed, is considered a runtime-error. This makes it possible to construct functions and procedures which accept a variable number of arguments, if a leading argument determines the number of arguments to follow (just like `printf()`).

Functions and procedures can be used recursively (with a finite depth of the calling stack, however).

As an example, the following defines a recursive implementation of the factorial function in terms of the hoc language:

```
func fac()
     {
     if ($1 <= 0)
        return 1;
     else
        return $1 * fac($1-1);
     }
```

There are four predefined procedures in YODA:

init(): init() is called exactly once if the task is loaded.

analyze(): This task is called each time an event matches a selector.

finish(): It is called every time YODA stops, i.e. if `stop` is pressed or if EOF is encountered.

refresh(): This task is associated with the display and NOT called in batch mode. Using the internal display of YODA, refresh() is called just before each timer triggered update of the display and if YODA stops. By default (see -z option), this is every 15 seconds.

Function	Purpose
`writevec(arrayName,dim,filename)`	write array of name `arrayName` and dimension `dim` to ASCII file with name `filename`.
`dumpvec(arrayName,dim,filename)`	write array of name `arrayName` and dimension `dim` to binary file with name `filename`.
`dumpvecShort(arrayName,dim,filename)`	same as `dumpvec(...)` except that double numbers are converted to short integers before writing (this saves disk space when applicable).
`readvec(arrayName,dim,filename)`	read ASCII file named `filename` and fill numbers into the array of name `arrayName` and dimension `dim`. Lines beginning with the `'#'` character are considered to be comments.
`sourcevec(arrayName,dim,filename)`	read binary file named `filename` and fill numbers into the array of name `arrayName` and dimension `dim`.
`writetext(textSelector,text_length, filename,filemode)`	writes text event selected by `textSelector` to file `filename`. The `mode` may be `"w"` for a new file or `"a"` for append. example: `select TEXT ;` `writetext(TEXT,TEXT_LENGTH,"test","w");`

Table 5.5: List of Functions for Input and Output of Arrays to Files

5.4 Input- and Output Functions

Variables can be assigned interactively with the `read(varname)` function. A number is expected to be typed in the terminal window, and — if valid — is assigned to the variable `varname`. The function `read_string()`does the same for strings, but is only allowed as an argument to other functions.

Output to the controlling terminal can be made with the familiar `printf(format, ...)` call. It behaves just like in "C", see e.g. `man printf` for details.

Input and Output of arrays or text-events to files can be accomplished by the functions listed in 5.3.

5.5 Histogramming Functions

The histogramming functions are summarized in Table 5.5. Histograms are created with calls to either `histo(...)` or `histo2d(...)` for one- or two-dimensional histograms, respectively. These calls return a *handle* to the created histogram which must be used to refer to the object when filling, clearing etc. This handle is used just inside the current task context and has nothing to do with e.g. PAW's histogram identifiers.

Function	Purpose
`hid = histo(name,nx,xmin,xmax)`	create histogram with name `name`, `nx` channels between `xmin` and `xmax`. return histogram id `hid`
`hid = histo(name,other_hid)`	create histogram with name by copying histo with id `other_hid`
`hid = hrebin(name,other_hid,nx)`	create histogram with name as a rebinned version (`nx` bins) of `other_hid`
`hfill(hid,x[,incr])`	increment histogram `hid` at value `x` by 1 [or `incr`]
`cont = hpoke(hid,bin,val)`	set contents of channel number `bin` to `val` for histogram `hid`. returns previous contents
`cont = hpokeErr(hid,bin,val)`	set error of channel number `bin` to `val` for histogram `hid`. returns previous error
`cont = hpeek(hid,bin)`	return current contents of channel `bin` for histogram `hid`
`cont = hpeekErr(hid,bin)`	return current error of channel `bin` for histogram `hid`
`val=hintegral2d(hid[,x1,x2][,err])`	calculates integral [over specified range of x values], [and stores the associated error in `err[0]`, i.e. err must be dim'ed as an array]
`hdump(hid)`	print contents of histogram `hid` to stdout
`hclear(hid)`	set all channels of histogram `hid` to zero
`hid = histo2d(name,nx,xmin,xmax, ny,ymin,ymax)`	create 2-dimensional histogram with name, and nx(ny) bins in the horizontal (vertical direction covering the range xmin-xmax (ymin-ymax)
`hid = histo2d(name,other_hid)`	create 2d-histogram with name by copying histo with id `other_hid`
`hid = hrebin2d(name,other_hid,nx,ny [,xmin,xmax,ymin,ymax])`	create histogram with name as a rebinned version (`nx` and `ny` bins) of `other_hid`, if desired only in a bin range specified by `xmin`, `xmax`, `ymin`, and `ymax`. Returns histogram identifier `hid`
`hfill2d(hid, x, y[, incr])`	increment histogram `hid` at value (`x`,`y`) by 1 [or `incr`]
`cont = hpoke2d(hid,binx,biny,val)`	set contents of channel number (`binx` , `biny`) to val and return previous contents
`err = hpokeErr2d(hid,binx,biny,val)`	set error of specified channel and return old value
`hpeek2d(hid, binx, biny)`	return current contents of channel (`binx`, `biny`) for 2-dimensional histogram `hid`
`hpeekErr2d(hid, binx, biny)`	return current error of given channel
`val=hintegral2d(hid[,x1,x2,y1,y2] [,err])`	calculates integral [if desired only over specified range of x and y values], [and stores the associated error in `err[0]`, i.e. err must be dim'ed as an array]
`hdump2d(hid)`	print contents of 2-dimensional histogram `hid` to stdout
`hclear2d(hid)`	set all channels of 2-dimensional histogram `hid` to zero

Table 5.6: List of Available Histogramming Functions

Function	Purpose
`value=hstat(hid,"dim")`	returns dimension of histogram
`value=hstat(hid,"id")`	returns HBOOK id of histogram
`value=hstat(hid,"binx")`	returns number of bins in x
`value=hstat(hid,"minx")`	returns upper limit in x
`value=hstat(hid,"maxx")`	returns lower limit in x
`value=hstat(hid,"binwidthx")`	returns width of bin in x
`value=hstat(hid,"biny")`	returns number of bins in y (0 for 1-dim)
`value=hstat(hid,"miny")`	returns upper limit in y (0 for 1-dim)
`value=hstat(hid,"maxy")`	returns lower limit in y (0 for 1-dim)
`value=hstat(hid,"binwidthy")`	returns width of bin in y
`value=hstat(hid,"maximum")`	returns maximum
`value=hstat(hid,"minimum")`	returns minimum
`value=hstat(hid,"overx")`	returns number of overflows in x
`value=hstat(hid,"underx")`	returns number of underflows in x
`value=hstat(hid,"overy")`	returns number of overflows in y
`value=hstat(hid,"undery")`	returns number of underflows in t
`value=hstat(hid,"error")`	returns 1 if errors are booked, else 0
`value=hstat(hid,"meanContents" [,x1,x2[,y1,y2]][,"var"])`	returns (weighted) mean of the histogram contents, x1(y1) and x2(y2) allow to specify a bin range in x(y), if a variable name is given in "var" the error of the mean is stored in this variable
`value=hstat(hid,"mean"[,x1,x2])`	returns the mean value of the 1-dim histogram [for bins x1-x2].
`value=hstat(hid,"rms",[,x1,x2])`	returns the root mean square value of the 1-dim histogram [for bins x1-x2].
`hname(hid[,1])`	returns pointer to histogram name (can only be used as an argument, e.g. to sprintf(...)., if 1 is specified as the second argument the task name is stripped off
`hmanip("add",hid1,hid2)`	performs `hid1 = hid1 + hid2`
`hmanip("sub",hid1,hid2)`	performs `hid1 = hid1 - hid2`
`hmanip("mul",hid1,hid2)`	performs `hid1 = hid1 * hid2`
`hmanip("div",hid1,hid2)`	performs `hid1 = hid1 / hid2`
`hmanip("addConst",hid,value[,error])`	performs `hid = hid + value`, if error is given, error propagation is done
`hmanip("subConst",hid,value[,error])`	performs `hid = hid - value`
`hmanip("mulConst",hid,value[,error])`	performs `hid = hid * value`
`hmanip("divConst",hid,value[,error])`	performs `hid = hid / value`
`hmanip("mulRows",hid1_2d,hid2_1d)`	performs `hid1_2d(i,j) *= hid2_1d(j)`
`hmanip("divRows",hid1_2d,hid2_1d)`	performs `hid1_2d(i,j) /= hid2_1d(j)`
`hmanip("mulCols",hid1_2d,hid2_1d)`	performs `hid1_2d(i,j) *= hid2_1d(i)`
`hmanip("divCols",hid1_2d,hid2_1d)`	performs `hid1_2d(i,j) *= hid2_1d(i)`
`hmanip("asymmetry",hid1,hid2,hid3)`	performs `hid1=(hid2-hid3)/(hid2+hid3)`
`hmanip("sqrt",hid)`	performs `hid = sqrt(hid)`
`hmanip("inverse",hid)`	performs `hid = 1/hid`
`hmanip("poisson",hid)`	calculate error of bin as sqrt of bin contents
`hmanip("xtitle", histoid, "Title")`	set title for x-axis in YODAs display
`hmanip("ytitle", histoid, "Title")`	set title for y-axis in YODAs display
`hdelete(hid)`	deletes histogram from memory

Table 5.7: List of Available Histogramming Functions (cont'd)

Function	Purpose
`hid_1d=hmean("new_name",hid_2d, "meanX"[,y1,y2])`	creates 1d-Histogram with name hid_1d "new_name" containing the mean and rms (as contents and error) of vertical slices of 2d-Histogram hid_2d. If y1 and y2 are given the mean and rms is calculated for this bin range
`hid_1d=hmean("new_name",hid_2d, "meanY"[,y1,y2] )`	the same but for horizontal slices
`hid_1d=hmean("new_name",hid_2d, "rmsX"[,x1,x2] )`	here the root mean square of vertical slices is stored as the histogram contents of hid_1d.
`hid_1d=hmean("new_name",hid_2d, "rmsY"[,y1,y2] )`	the same but for horizontal slices
`update()`	transfer current contents of all histograms to shared memory (for the displaying program) explicitly. Periodic update can be stopped with `set_updatetimer 0` in YODA's terminal window
`write_histos(filename [, format])`	writes all histograms of all tasks to file named `filename` in format HBOOK (0) or YZ (1)
`clearhistos()`	clears all histograms of all tasks.
Read Histograms by name:	
`file = hfile("filename"[, 1])`	open HBOOK file [strip taskname from histogram names]
`hclose(file)`	close HBOOK file
`n = hlist(file)`	list contents, returns number of histos
`hid = hrin(file,dim,"name" [,"new_name"])`	read histo of dimension dim from file [and change name]
`n=hrinall(file,vector,dim[,"ext"])`	read at most dim histos, ids will be in vector, returns number histos read. If `ext` is given, histogram names will be "taskname.ext.name-in-file"
Read Histograms by ID:	
`hid   = hread("file",id[,"name"])`	read histogram of any dimension [and change name]
`hid1d = hread1d("file",id[,"name"])`	read only 1d histograms
`hid2d = hread2d("file",id[,"name"])`	read only 2d histograms
projections and slices:	
`hid1d=hprox(name,hid2d,ybin1,ybin2)`	creates 1-dimensional histogram `hid1d` as a slice of the 2-dimensional histogram `hid2d` using bins `ybin1` to `ybin2` in y. If `ybin2` is ommitted only one bin is used. If both `ybin1` and `ybin2` are not given, all bins in y are used for the projection.
`hid1d=hproy(name,hid2d,ybin1,ybin2)`	creates 1-dimensional histogram `hid1d` as a slice of the 2-dimensional histogram `hid2d` using bins `xbin1` to `xbin2` in x. If `xbin2` is ommitted only one bin is used. If both `xbin1` and `xbin2` are not given, all bins in x are used for the projection.

Table 5.8: List of Available Histogramming Functions (cont'd)

ERRNO value	meaning
2	file does not exist
12	creation of object failed
14	histogram/file not found
17	dublicate name
22	invalid arguments

Table 5.9: List of error codes used by histogramming functions

The usual way of using histograms is by means of *filling* via calls to `hfill` or `hfill2d`, respectively. However, you can also "abuse" the histograms for producing x/y plots, where one quantity is simply plotted against another. Simply use `hpoke(h, bin, y);` for entering a point at (bin,y). It is important to remember to use the x-coordinate in units of bins (indexing with floats will not give the desired results). Once all desired channels are filled, the information can automatically be transferred to the display program with the `update();` call. This causes the current contents of the histograms to be transferred to paw (To make this work correctly, you should switch off the automatic (periodic) update done by YODA by default. To do so, enter `set_updatetimer 0` in YODA's terminal window. Periodic update can be restored with `set_updatetimer VALUE`, where `VALUE` is the time between refresh cycles in milliseconds). The histogram can then be cleared with `hclear(h);`. The corresponding calls for two-dimensional histograms are available as well and work just in the same manner.

Histograms can also be read from and written to file in either HBOOK or YODAs native YZ format (see table 5.5).

The histogramming functions set a variable ERRNO, to test if the last command was successful (ERRNO == 0) or not (ERRNO > 0). See table 5.5 for an explanation of the error codes.

5.6 Using Cuts

Function	Purpose
`cid = cut(name, initial)`	create cut with name `name` and default value `initial`. returns cut identifier `cid`
`inside(cid, x)`	checks whether `x` is inside the limits of cut `cid`
`cid = cut2d(name, initial)`	create 2-dimensional cut with name `name` and default value `initial`. returns cut identifier `cid`
`inside2d()`	checks whether `x` is inside the limits of 2-dimensional cut `cid`
`readcuts(filename)`	read cut parameters from file `filename`

Table 5.10: List of Available Cut Functions

To perform typical physical analysis, the possibility for *cuts* must exist. In YODA, there are one- and two-dimensional cuts (corresponding to one-and two-dimensional histograms). Cuts need not be associated to histograms, but very often will. The usage of cuts is much like the use of histograms: They are created with a single call returning a *cut handle*, and later referred to by this handle.

5.6.1 One-Dimensional Cuts

One-dimensional cuts reflect the simple criterion of a value being in a certain numerical range, e.g. "is the current value of the variable x higher than 0.5 and lower than 0.8 ?". This can of course be handled "by hand", e.g.

```
if ( x > 0.5 && x < 0.8)
   printf("Value is in range");
```

,but with the use of cuts the analysis becomes much more readable. Cut parameters (e.g. upper and lower limit in the previous example) are stored in files for archiving and later retrieval. Furthermore, PAW++ allows to change the cut limits graphically (well, with some effort...).

The creation of one-dimensional cuts is done with the call `cut(name,initial);`. `name` is used to identify the cut to the "outside world", i.e. PAW. Note, that due to restrictions in PAW's handling of cuts, the names must be numbers, although this is not required (and desired) from YODA. The second parameter is the default logical value that the cut should have, as long as no points for the cut are defined. You can choose either "true" (1) , meaning that the cut always holds (i.e. all values are inside the cut limits), or "false" (0), meaning the cut never holds (i.e. all values are outside the cut limits).

So a typical call to create a 1-dimensional cut would be:

```
proc init()
{
/* create the pulse height cut with id 30. default value is true (1) */
phcut = cut("30", 1);
}
```

Upper and lower limit are not set yet for this cut. To use the cut in `proc analyze()`, use the `inside(CUT,X)` call. For instance, to reproduce the example above, you would write:

```
if (inside(phcut, x))
   printf("Value is in range");
```

To read cuts from a file, you can either choose the hoc-function `readcuts(FILENAME)` to read the cut parameters from file `FILENAME`, or you can read or save them them via the YODA cut pulldown menu.

To produce such a file, you (currently) have to "cheat" PAW++. Since inside PAW++, cuts have only a meaning in the context of *ntuples* (which will not be discussed here), you must pretend to define a cut for an ntuple. By default, cuts are 1-dimensional inside PAW++. Simply draw your histogram inside PAW++ and enter:

```
cut 30 g
```

Then, you can click inside the histogram with the left mouse button to define lower and upper limit (in x) of the cut. Finish the definition with the left mouse button. You can then draw a different histogram, start a new cut definition with `cut IDENTIFIER g` and so on. To use these cuts again inside YODA, they must have the same `IDENTIFIER` as the cut's name in the user task. This is the reason for using only numerical "names" for the cuts inside the user tasks.

After all cuts are defined, they can be written to file with the PAW command `cut 0 w FILENAME`, where `0` means "all cuts" and `FILENAME` is the name of the file where

PAW should save the cut parameters. (Although it is an ugly format, it is plain ASCII, and you can edit it afterwards with an editor, if you want to).

This file can then be read into YODA either via the "cut" pull-down-menu or inside the user task (`proc init()` with the call `readcuts(FILENAME);` . By doing this, the setting of the cut parameters become effective immediately.

5.6.2 Two-dimensional Cuts

Analogous to one-dimensional cuts, two-dimensional cut mean the condition of a *pair of values* being in a certain range, where the range is defined by a closed polygon in the x/y space. The condition is then e.g. "is the current point (x,y) inside the area defined by the polygon I just drew into the picture representing the whole (x/y) space ?". This is no longer done easily by hand, as in the case of one-dimensional cuts.

Creating a two-dimensional cut is done with the call `cutid = cut2d(NAME, initial);`, with the parameters just as for one-dimensional cuts. Again, the handle (`cutid` returned is used to refer to the cut later.

Inside `proc analyze()`, the cuts are used with e.g.:

```
if (inside2d(cutid, x,y))
   printf("%f %f is inside the cut!\n", x, y);
else
   printf("%f %f is outside the cut!\n", x, y);
```

Normally, you would use histogramming functions instead of printing the values. This is done simply for illustration.

As their one-dimensional counterparts, cuts are read from file either via the cut pull-down menu or with the `readcuts(FILENAME)` call, which works for one- and two-dimensional cuts as well. In fact, cut parameters can be mixed in the same file (as it is done by PAW).

To define a two-dimensional cut graphically, you must first convince PAW that you've just plotted a two-dimensional representation of some ntuple variables (else it will still stick to the opinion that you are viewing a one-dimensional histogram, and will not permit to enter more than two points ...).

To do this trick, use the following sequence of paw kumac commands:

```
nt/create 9999 'JUNK' 2 ! 1000 x y
nt/pl 9999.x%y
h/del 9999
h/del 1000000
next
```

To make this more convenient, you should save this sequence in a kumac-file for easy retrieval. Call it `cut2d.kumac` or something like this, so that you can use it anytime you want to define 2-dimensional cuts with PAW.

If you want to switch between one- and two-dimensional cut definition in PAW, you should write another kumac file to convince PAW of the idea of the necessity to allow one-dimensional cuts only:

```
nt/create 9999 'JUNK' 2 ! 1000 x y
nt/pl 9999.x
h/del 9999
h/del 1000000
next
```

Save this in another kumac-file, e.g. `cut1d.kumac`, so switching between one- and two-dimensional definitions is easy.

5.7 Communication between Hard and Soft Callbacks

To access or change variables which are defined in *hard callbacks* (cf. chapter 7) a so-called Broker is used. These variables may be double values, arrays of double values or strings and the corresponding functions are listed in table 5.7. These "Broker-variables" are accessed through their names. The name of variable `anExample` in the hard callback named `veryHardCallback` would be: `"veryHardCallback:anExample"`.

Furthermore, hard callbacks may be enabled or disabled by soft callbacks.

Function	Purpose
`x = broker(double_name)`	assign to `x` the value of the broker variable named `double_name`
`x = broker(array_name)`	assign to `x` the dimension of the broker variable named `array_name`
`broker(double_name,value)`	assign `value` to the broker variable named `double_name`.
`broker(array_name,array,dim)`	copy `array` of dimension `dim` to the broker variable named `array_name`.
`broker(string_name,string)`	assign the `string` to the broker variable named `string_name`.
`enable(HardCallbackName)`	enable hard callback
`disable(HardCallbackName)`	disable hard callback

Table 5.11: List of Functions for Communcation between Hard and Soft Callbacks

Chapter 6

YODA tcl Commands

YODA graphical user interface (GUI) is written with the tcl/tk windowing toolkit. Much of YODAs funktionality is implemented in tcl commands which may be accessed through the GUI or directly by typing the appropriate command in YODAs main window. The output will appear on YODAs main window and may prove useful for logging purposes. Aditionally the tclCommand() builtin function allows to execute arbitrary tcl commands from within the hoc language.

These commands may be included in the `yoda.rc` file which is executed on YODA startup.

Function	Purpose
`clearhistos`	clears all histograms
`clearhisto` *histoname*	clears histogram *histoname*
`dumphisto` *histoname*	dump histogram contents of *histoname*
`histowrite`*file* `[ mode ]`	write all histograms to file, if mode=0(1) is given force output in HBOOK (YZ) format
`projecthisto` *name d b*	creates projection or slices for histogram with *name* in *d*=x,y direction. If *bin* is 0 a projection is generated, otherwise slices of *b* bins are created.
`setPawRzfileMode` *value*	if value="rz" ("yz") the default storage type for histograms will be HBOOK(YZ)
`.display.sh clear`	clear display
`.display.sh addInc` *name*	display histogram in new zone
`.display.sh add` *name*	overlay histogram in last zone
`.display.sh sub` *name*	remove histogram in all zones
`.display.sh title` *a b*	if a(b) is 1 show histogram (axis) titles
`.display.sh zone` *x y*	set number of rows (y) and columns (x) in display
`Print2File`	writes current display to file `zoni.ps`

Table 6.1: List of useful tcl commands (histogramming and display)

Function	Purpose
`offline`	puts YODA offline, i.e. data sorting is suspended
`online`	puts YODA online, i.e. data sorting is continued
`close_input`	close input stream
`open_inputfile` *filename*	open input file and attach to YODAs inpur event stream
`open_inputproc` *input_process*	open input process and attach to YODAs inpur event stream
`disable` *callbackName*	disable soft or hard callback
`enable` *callbackName*	enable soft or hard callback
`load` *hocfileName*	load a hoc file
`unload` *callbackName*	unload a hoc file
`reload` *callbackName*	reload a hoc file
`get_brokerinfo_long_sorted`	dump all broker values and their values
`get_cutinfo`	get information on cuts
`get_eventstack`	dump info on all subevents encountered by YODA so far
`get_histoinfo`	get information on histograms: entries, overflows etc.
`get_runinfo`	get information on runnumber, #events, processing speed etc.
`get_statusstr`	get information on input and output streams
`get_taskinfo`	get information on Callbacks, # called, accepted etc
`get_versionstr`	get YODA version
`set_broker` *variable value*	set broker variable
`set_containermode` *on/off*	if "on" container subtypes are considered in subevent selection (default "off")
`set_displaytimer` *value*	set update time interval (like `yoda -z`)
`set_updatetimer` *value*	set update time interval (like `yoda -t`)

Table 6.2: List of useful tcl commands

Chapter 7

Beyond the Hoc Language

As mentioned in the introduction, the abilities of the hoc language are sufficient for many typical tasks in physical analysis. There are, however, special cases for which the speed and/or the interpreter language capabilities are not sufficient.

For these cases, there are two ways of enhancing YODA's abilities while mainly retaining the comfortable features.

- One way is to add new functions to the set of built-in hoc library routines. After adding own, customized routines, they are indistinguishable from the "native" hoc functions. They can be used to perform lengthy calculations at full, compiled speed or memory-greedy algorithms with complex datastructures.. Since they are written in C (C++), they may use the full spectrum of the language and all libraries available. See section 7.1 on how to to add new functions to the hoc language. The interface to and from these functions is still restricted by the hoc language, however.

- Introduce own *hard callbacks* into YODA. This means to write one or more complete C++ classes (derived from a basecall providing the framework), which perform arbitrary operations on the incoming data. There is no restriction on how to operate on the data, and the selection criteria (when to process and when to ignore data) is completely in the user's code. On the other hand, YODA' principle framework is preserved. These callbacks have full access to YODA's histograms and cuts, and their appearance to the user is like a user task (which cannot be loaded and unloaded, however). See section 7.2 on how to implement such a class.

7.1 Enhancing the Hoc Language

Enhancing the hoc language is in fact very easy. First, you need access to YODA's source code. Then, you should write your new hoc builtin-function according a certain template. Suppose you want to add a new function, which gets two parameters, adds them, and returns the result (a slow way of saying `a+b`...). Your implementation could look like:

```
/*>-----------------------------------------------------------------------------*/
double add(int i, Datum *p)
/*
 * add its two parameters
 *
 * input: i: number of parameters actually passed
 *        p: the arguments
 *
 * return: sum of argument 1 and argument 2
 *         -1: wrong number of arguments (not too clean...)
```

```
 */
/*<-------------------------------------------------------------------------*/
{
if (i != 2)
   {
   cerr << ((i<2) ? "too few arguments\n" : "too many arguments\n");
   return -1;
   }

return p[0].val + p[1].val;
}
```

Parameters are passed as an array of type `Datum`, which is a union to hold all types of objects from within the user-task. Its definition is:

```
union Datum
{                                /* interpreter stack type */
Symbol  *sym;
char    *str;
double  val;
};
```

For user applications, only the `str` and `val` components are relevant. whenever an arithmetic value (e.g. a variable or expression) is passed, `val` is used to access the actual value. If the function is called with a string argument (remember: there are only string literals inside hoc), the `str` component is used.

To make this function part of the built-in set of hoc functions, you have to add it to the structure `builtins_va[]` in `init.cc`:

```
...
extern double add(int i, Datum *p);  /* declare your function */
...
static struct
{               /* Built-ins with varargs */
char    *name;
double  (*func)(int, Datum *);
} builtins_va[] =
{
{   "printf", print},
{   "histo", histo},
{   "hfill", hfill},
...
{   "update", update},
{   "add",    add},        /* <- your new function in the list */
{   0,        0  }
};
```

Compile, and check out your new built-in with a simple hoc file like:

```
proc init()
     {
```

```
    c = add(1.234, 4.321);
    printf("sum is : %f\n", c);
    }
...
```

7.2 Implementing Hard Callbacks

Implementing hard callbacks is a bit harder than enhancing the hoc language, but offers utmost flexibility in handling the data. You should know something about the C++ programming language, see [str93]. Hard callbacks behave much like the callback mechanism from the X-window system. The idea is not to actively process all data, but to "get called" by the main program as a sort of interrupt routine.

Every hard callback is a implemented as a C++ class, derived from a baseclass `HardCallback`. This baseclass provides the framework, in which callback's functions must be written. Study `HardCallback.h` and `HardCallback.cc` for details. Each callback "lives" in association with YODA's `Eventstream` class. This class is the central device for reading, writing and dispatching containerevents and subevents. See `Eventstream.h` and `Eventstream.cc` for details.

Hard callbacks can access data of and add subevents to the incoming eventstream. The user must specify the following:

- the virtual function `init()` which is called at any time a variable managed by the `broker` utility has changed.
- the virtual function `analyze()`, called for every accepted event and does the actual data analysis.
- the virtual function `finish()`. This function is called whenever YODA is going offline.
- the subevents to be selected or accessed (same meaning as in the `hoc` language.
- the subevents which are created by the hard callback and should be added to the eventstream after the analysis.

7.2.1 A Sample Hard Callback

Here is a sample hard callback:

```
#ifndef USER_HARD_TEST
#define USER_HARD_TEST
/*>-----------------------------------------------------------------------*/
/*
 * HardTest.h
 *
 * userdefined class derived from class HardCallback base class
 */
/*<-----------------------------------------------------------------------*/

#include "HardCallback.h"

class HardTest : public HardCallback
{
 private:
SelectedData<uint32> adc1;
SelectedData<uint32> adc2;
AccessedData<uint32> tdc;
GeneratedData<double> adcsum;

// variables managed by Broker
char *gainfile;
double globalgain;
double *tdc_offset;

//other variables
double gain[10];

 public:

HardTest(const char*);
~HardTest();

virtual int init();
virtual int analyze();
virtual int finish();
virtual const char *help();
};

#endif
```

Its implementation could look like:

```
/*>-------------------------------------------------------------------------*/
/*
 * HardTest.cc
 *
 * implementation of class HardTest
 */
/*<--------------------------------------------------------------------------*/
#include <string.h>
#include <stdio.h>
#include "HardTest.h"

static const char *helpstring =
   "this is HardTest, a sample hard callback!\n"
   "here should be a description on what HardTest is doing!\n";

/*>--------------------------------------------------------------------------*/
HardTest::HardTest(const char *n) : HardCallback(n,1)
/*
 * constructor, the second argument (here: 1) is the priority of the
 * HardCallback (the callbacks are processed in the order with
 * increasing prioity number)
 */
/*<--------------------------------------------------------------------------*/
{
// initialize variables managed by broker
global_gain=10;
gainfile = new char[strlen("dummy")+1];
strcpy(gainfile,dummy);
time_offset = new double[10];
for (int i=0 ; i<10 ; i++)
   time_offset[i] = (double) i;

// link subevents:
selectInput( "ADC1"  , adc1.instance()  , DataType::UNSIGNED_LONG);
selectInput( "ADC2"  , adc2.instance()  , DataType::UNSIGNED_LONG);
selectInput( "TDC"   , tdc.instance()   , DataType::UNSIGNED_LONG);
selectOutput("ADCSUM", adcsum.instance(), DataType::DOUBLE);

// make variables known to the broker
announce("globalgain","global gain factor (for all channels)",globalgain);
announce("gainname","path to file containing gains",gainfile);
announce("time_offset","time offset for our TDCs", time_offset,10);
init();
}

/*>--------------------------------------------------------------------------*/
HardTest::~HardTest()
/*<--------------------------------------------------------------------------*/
{
delete[] gainfile;
delete[] time_offset;
}

/*>--------------------------------------------------------------------------*/
virtual int HardTest::init()
/*
 * called by broker, when a variable has been set
```

```
 */
/*<----------------------------------------------------------------------------*/
{
// read gains
FILE *fp=fopen(gainfile,"r");
double *tmp = gain;
while( fscanf(fp,"%lf",tmp) == 1)
      tmp++;
}

/*>----------------------------------------------------------------------------*/
virtual int HardTest::analyze()
/*
 * called for every event which matches the selected subevents
 * return 1 on success
 */
/*<----------------------------------------------------------------------------*/
{
accepted++; // this is for YODAs GUI: count number of accepted events

// do anything you like
for (int i=0 ; i < adc1.get_length() ; i++)
   if ( tdc[i] - time_offset[i] > 777)
       adcsum[i] = globalgain*(adc1[i] + adc2[i]*gain[i])
   else
       adcsum[i]=0;

adcsum.set_length(adc1.get_length());

write_data(); // write all GenereratedData to existing eventstream

return 1;
}

/*>----------------------------------------------------------------------------*/
virtual int HardTest::finish()
/*
 * called when YODA goes offline
 */
/*<----------------------------------------------------------------------------*/
{
// do some nonsens
printf("HardTest: used global gain %f\n",globalgain);
return 1;
}




/*>----------------------------------------------------------------------------*/
virtual const char *HardTest::help()
/*
 * return a string which tells the user what the hard callback does
 * this text fills the textbox in YODAs GUI when help on this hard
 * callback is requested.
 */
/*<----------------------------------------------------------------------------*/
{
```

```
return helpstring;
}
```

Finally the existence of our hard callback must be announced to YODAs `Eventstream` class in the `user.cc` file:

```
/*>------------------------------------------------------------------------*/
/*
 * user.cc
 *
 * userdefined classes derived from class HardCallback
 *
 */
/*<------------------------------------------------------------------------*/
#include <eventstream.h>
#include "HardTest.h"

// now instanciate all used classes (needed for GNUs gcc compiler!)
#ifdef __GNUC__
template class SelectedData<uint32>;
template class AccessedData<uint32>;
template class GeneratedData<double>;
#endif

void user()
{
Eventstream::instance().link(new  HardTest("HardTest"));
}
```

7.2.2 Subevent Handling

There are predefined classes to handle subevents:

`SelectedData<T> name;`	defines `name` to be a variable linked to an existing subevent (as `select` in the hoc language).
`AccessedData<T> name;`	like `access` in the hoc language
`GeneratedData<T> name;`	`name` identifies a subevent to be added to the current containerevent.

where `T` is a storage type. Supported storage types are listed in table

The most important member functions of your callback will be the virtual functions `init()`, `analyze()`, and – if desired – `finish()`.

The function `accept()` is called by YODA (by the `eventstream` object) once for each containerevent. Via the member functions of `Eventstream`, the callback has full access to the containerevent's data. It can then decide, whether to accept (hence the name) the event for processing — it will return a 1 (one) — or to ignore this containerevent. It may also pick out one or more pointers to subevents to deal with later on.

If this function returns one, the member function `analyze()` will be called. Here the actual (complicated) analysis should take place. By using `Eventstream` member functions, all information about and inside the current containerevent can be queried. Moreover, subevents can be deleted and new subevents (containing results of the analysis) can be generated and added to the current containerevent.

With the optional member function `finish()`, results can be printed or saved whenever the data analysis is stopped (either by end-of-file or by user interaction).

As you may have noticed, the member functions `analyze()` and `finish()` correspond exactly to the hoc-procedures with the same name. The `accept()` member function, however, is hidden in the context of the hoc language and works according to the `select` statements given in the hoc procedure `init()`.

type in selector.db	name	C++ storage type	comment
undefined	UNDEFINED	any class	may be anything (caution!)
char	SIGNED_CHAR	char	
uchar	UNSIGNED_CHAR	unsigned char	
short	SIGNED_SHORT	short	
ushort	UNSIGNED_SHORT	unsigned short	
long	SIGNED_LONG	int32	
ulong	UNSIGNED_LONG	uint32	
int8	SIGNED_INT8	int8	
uint8	UNSIGNED_INT8	uint8	
int16	SIGNED_INT16	int16	
uint16	UNSIGNED_INT16	uint16	
int32	SIGNED_INT32	int32	
uint32	UNSIGNED_INT32	uint32	
int64	SIGNED_INT64	int64	only supported on
uint64	UNSIGNED_INT64	uint64	64bit architecture
float	FLOAT	float	
double	DOUBLE	double	

Table 7.1: Supported data types for subevents

After building your hard callback class, you must create a single object of this class (instantiate it), and "announce" this callback to YODA's `eventstream` object. This is done in `user.cc` with the call `eventstream`'s member function `link()`.

7.3 Writing Data with YODA

To write data with YODA, you can simply use the `write_data()` function of the `hoc`-language or the `put()` member function of the `Eventstream` class (when called from a hardcallback). Once you use this function, the current contents of the eventstream (a "snapshot" containing one complete containerevent) is written to the current output file. Output to file can be specified either on YODA's commandline (output option), or interactively with the `File` pulldown menu (some minor modifications in the graphical user interface are required).

The call to `put()` or `write_data()` writes the current contents of the containerevent to the output file. This includes all subevents, which may have been added by the hard callback(s) defined. All deleted subevents will disappear in teh output as well.

Be sure to use the `put()` member-function or `write_data()` function only once for each containerevent, i.e. from one callback only, else you will get multiple containerevents with the same contents into the output file. If you have multiple callbacks, which add or delete subevents independently you should probably add the `put()` call into `Eventstream`'s member function `dispatch()` only.

Appendix A

Writing YODA-Style Data with Your Own Program

You may be forced to write YODA-style data output (e.g. from a simulation program or an input formatter). This should be no problem using the `Eventstream` class. See the example below:

```
/*>----------------------------------------------------------------------------*/
/*
 * a short program to test the ability of producing containerevents
 */
/*<----------------------------------------------------------------------------*/

#include "eventstream.h"

/*>----------------------------------------------------------------------------*/
main()
/*
 *
 */
/*<----------------------------------------------------------------------------*/
{
/*
 * declare the application-specific data to be written
 */
struct Mydata                   // some nonsens we want to write
   {
   double foo;
   float  bar;
   int    stuff;
   } data;

/*
 * now make a new eventstream, which pumps the containerevents from
 * filedescriptor IN to file OUT. It is also used to generate containervents
 * from scratch (this is used here), and then write them to file.
 * default is stdin / stdout
 */

Eventstream eventstream(stdin, stdout); // this defaults to stdin / out

/*
 * setup a new containerevent. this is needed everytime the containerevent
```

```
 * structure is modified (NOT regarding the eventnumber.
 * choose type/subtype according to your needs. variable names below are used
 * just for verbosity
 */
short triggerpat=1;
short runnum=12;
short expnum=42;
unsigned evtnum=0;

eventstream.setup(CEVT_T_SIMULATION, triggerpat, runnum, expnum, evtnum);

/*
 * now produce 100 containerevents in a loop. first calculate
 * the quantities to be written, then add thenm to the current containerevent
 * as a subevent
 */

for (int i=0; i<100; i++)
    {
    data.foo = sin(i);
    data.bar = cos(i);
    data.stuff = random();

    /*
     * now add this as a subevent. supply type/subtype, pointer to data
     * and length of data.
     */

    eventstream.add_subevent(SEVT_T_CALCULATED, SEVT_ST_DUMMY,
                             &data, sizeof(data));
    cerr << "writing container number " << i << "\n";
    eventstream.put();          // write to output
    eventstream.next_container(); // increment the event number
    }
}
```

Bibliography

[ker84] Brian Kernighan, Rob Pike: The Unix Programming Environment, Prentice Hall 1984

[sch94] Florian Schwandt: TDAS ISKP User's Guide, Bonn University, ISKP internal report 1994

[str93] Bjarne Stroustrup: The C++ Programming Language, Addison-Wesley 1993