SAL: a Symbolic Analysis Language

Gratefulfrog: gf_at_gratefulfrog_dot_net

August 23, 2006

Contents

Abstract

This document is a literate program called SAL: *Symbolic Analysis Language.* SAL runs in the Common Lisp environment. SAL has been tested in both on GNU CL and CMUCL.

The support for the Literate version of SAL is based on noweb, LaTeX, TeXand their friends. All those systems are needed to be able to build and install SAL.

The idea behind literate programming is that a program should *read like a book* if it is to be maintainable over time. The use of noweb enables us to maintain a single set of *source* files containing both description and code. These files are manipulated by the noweb functions to extract code and documentation. This is often called: "*tangling* out the code" and "*weaving* up the documentation."

In this version of SAL, the GNU make utility is used to perform all the tangling and weaving so that understanding the exact details of the commands used is not needed by the user who simply wants to build and install SAL in the Common Lisp environment.

Chapter 1

License

GPL http://www.gnu.org/licenses/gpl.html

Chapter 2

Quick Start

If you're in a hurry to get this running then:

Configure SAL: chapter 3 on page 3,

Make-Install: chapter 5 on page 32.

Chapter 3

User Configuration

This chapter contains the user configuration parameters for SAL. Before building SAL, these values must be set in the *noweb* source file `user-config.nw`. When updating:

- first make a backup copy of `user-config.nw`,
- be careful to *avoid adding extra blank lines* which could cause problems during file generation,
- be sure that all path names end in the slash "/" character,
- don't forget to make that back-up copy!

The following names *must* be set to appropriate values on your system.

The full path to the location of the configuration file `sal-config.lisp`. This is both a target location for **make install** and a reference used during SAL execution:

3a ⟨*configfile-path* 3a⟩≡ (34 53a)

```
/home/bob/.sal-config.lisp
```

The full path to the location of the lisp source files after they are built from the *noweb* sources:

3b ⟨*src-file-path* 3b⟩≡ (150a)

```
/home/bob/Desktop/Programming/lisp/StockEvaluator/Work/V7.0/
```

The full path to the location of the binary files. This is both a target location for **make install** and a reference used during SAL execution:

3c ⟨*bin-file-path* 3c⟩≡ (34 39c 150b)

```
/home/bob/Desktop/Programming/lisp/StockEvaluator/Work/bin/
```

The full path to the location of the model and rule files. This is a reference location used only during SAL execution:

3d ⟨*model-rule-path* 3d⟩≡ (34 151a)

```
/home/bob/Desktop/Programming/lisp/StockEvaluator/Work/bin/Examples/
```

The name of the model file that SAL will read during execution:

4a ⟨*mode-filename* 4a⟩≡ (151b)

```
model.lisp
```

The full path to the directory where SAL will write all output during execution:

4b ⟨*io-path* 4b⟩≡ (152a)

```
/home/bob/Desktop/Programming/lisp/StockEvaluator/Work/Output/
```

The name of the log file that SAL will write if the logging option is TRUE:

4c ⟨*log-filename* 4c⟩≡ (152b)

```
sal-log.out
```

The full path to the Lisp executable that will be used to compile and run SAL. This is only used by the `Makefile`. It is never used during SAL execution:

4d ⟨*lisp-path* 4d⟩≡ (34)

```
/opt/cmucl-19c/bin/lisp
```

Chapter 4

User Manual

4.1 Overview

The purpose of the Symbolic Analysis Language is to provide a flexible language and analysis engine enabling the application of rule-based data projection techniques to create and update a model of a set of data.

The platform maintains in memory a data set which can contain both intemporal, i.e. always true as well as time-dependent data.

Rules may be written as normal lisp functions which project/derive data values for specifically desired dates via so-called *sugar-functions* (chapter 4.3.5 on page 14).

All data is also available for query and/or update via the *sugar-functions* at the lisp command line, or via automated reporting as comma separated values either to a file or to std-out.

This document uses a stock market data-base as the basis for illustrations.

4.2 Quick Start

So, you want to see it work and don't want to know how, why, etc.? Well, so be it. But, if you never read the rest of the documentation you may never know about the amazing functionality that is too powerful to describe in this section[1].

4.2.1 Installation

Once you have updated the file *user-config.nw* (cf. chapter 3 on page 3), you should be able to follow the instructions on how to make and install SAL as provided in chapter 5 on page 32.

If it's too hard to follow those links, here's the short version:

[1] Not reading the documentation has been the reason that humanity has missed out on great things such as Fermat's theorem, learning that the Earth was round, and Frank's discovery of a limitless supply of free energy available to all.

5 ⟨*make-install-sal* 5⟩≡

```
$ make
$ make
$ make install
$ make sal-test
```

If that doesn't work, then you'll just have to read the above referenced pages.

4.2.2 System Execution

After the system has been successfully built and installed, it's time to try it out!

Did you install the examples? If not, do it now:

6a ⟨*install-examples* 6a⟩≡

```
$ make examples
```

Now you are ready to start the lisp environment:

6b ⟨*start-lisp* 6b⟩≡

```
$ lisp
CMU Common Lisp 19c (19C), running on bartbox
With core: /opt/cmucl-19c/lib/cmucl/lib/lisp.core
Dumped on: Thu, 2005-11-17 15:12:58+01:00 on lorien
See <http://www.cons.org/cmucl/> for support information.
Loaded subsystems:
    Python 1.1, target Intel x86
    CLOS based on Gerd's PCL 2004/04/14 03:32:47
*
```

Next, load the sal binary (be sure to use the correct path for your system):

6c ⟨*load-sal-bin* 6c⟩≡

```
* (load "/home/sal/bin/sal.x86f")
; Loading #P"/home/sal/bin/sal.x86f".
;; Loading #P"/home/sal/Config/sal-config.lisp".
;; Loading #P"/home/sal/bin/utilities.x86f".

... lots of information is output....
T
*
```

Then, load some data:

6d ⟨*load-test-data* 6d⟩≡

```
* (load "/home/sal/bin/Examples/data.lisp")
; Loading #P"/home/sal/bin/Examples/data.lisp".
T
*
```

Now, run the test harness and check the results!

7 ⟨*run-sal-test* 7⟩≡

```
* (wut:test "sal")
;;; Starting dribble.
;;; (wut::drib)
;;; Go for an analysis on [2005 2010],
      verbose,
      print to std-out
;;; (format t "~S~%" (sal:process :ticker-string "ibm"
                                   :current-year 2006
                                   :verbose t
                                   :data *ibm-data*
                                   :report-start 2005
                                   :report-stop 2010
                                   :std-out t
                                   :file-out t
                                   :out-file-name "this-is-output"
                                   :logging t
                                   :loop-detect t))
;
... Lots of info here, then ...
"Ticker","ID","Field",yr_2005,yr_2006,yr_2007,yr_2008,yr_2009,yr_2010
"ibm",4786,"Profit",83000.0,85000.0,85000.0,85000.0,100000.0,100000.0
"ibm",4786,"After Tax Earnings",7750.0,8250.0,8250.0,8250.0,10327.5,
10327.5
"ibm",4786,"Capital Expenditure",4030.0,4275.0,4275.0,4275.0,4995.0,
4995.0
"ibm",4786,"Smoothed Capital Expenditure",2.6133333333333333,
2.6999999999999997,2.766666666666667,2.85,3.1333333333333333,
3.4166666666666665
"ibm",4786,"Free Cash Flow",12662.386666666667,13162.3,
13162.233333333334,13162.15,15239.366666666667,15239.083333333334
"ibm",4786,"Employees",329001.0,329001.0,329001.0,329001.0,329001.0,
329001.0
"ibm",4786,"Manufactured Goop",100.0,100.0,100.0,100.0,100.0,100.0
"ibm",4786,"Manufactured Toto",2006.0,2007.0,2008.0,2009.0,2010.0,
2011.0
... More info, then ...

"Ticker","ID","Field",yr_2005,yr_2006,yr_2007,yr_2008,yr_2009,yr_2010
"ibm",4786,"Profit",83000.0,85000.0,85000.0,85000.0,100000.0,100000.0
"ibm",4786,"After Tax Earnings",7750.0,8250.0,8250.0,8250.0,10327.5,
10327.5
"ibm",4786,"Capital Expenditure",4030.0,4275.0,4275.0,4275.0,4995.0,
4995.0
"ibm",4786,"Smoothed Capital Expenditure",2.6133333333333333,
2.6999999999999997,2.766666666666667,2.85,3.1333333333333333,
3.4166666666666665
"ibm",4786,"Free Cash Flow",12662.386666666667,13162.3,
```

```
13162.233333333334,13162.15,15239.366666666667,15239.083333333334
"ibm",4786,"Employees",329001.0,329001.0,329001.0,329001.0,329001.0,
329001.0
"ibm",4786,"Manufactured Goop",100.0,100.0,100.0,100.0,100.0,100.0
"ibm",4786,"Manufactured Toto",2006.0,2007.0,2008.0,2009.0,2010.0,
2011.0
#<EQUAL hash table, 278 entries {586BA6ED}>
;;; Stopping dribble.
;;; (wut::drib nil)
; Evaluation took:
;   1.13 seconds of real time
;   0.696894 seconds of user run time
;   0.074989 seconds of system run time
;   1,135,577,868 CPU cycles
;   [Run times include 0.13 seconds GC run time]
;   1 page fault and
;   40,281,672 bytes consed.
;
T
*
```

So you've made SAL run! But what if this failed? What should you do? Well, you've got all the lisp source files, you've got the test data, you've got the Makefile, and you're reading the full system documentation. What more do you need?

This ends our quick-start. Those who want to fully understand should read on, others can stop here...

4.3 How-To's

4.3.1 How to Understand SAL

The purpose of SAL is to provide an infrastructure that can deduce new values from old ones. For example, if SAL knows that the sun rose *today* and if SAL is given a rule:
If the sun rose yesterday, then it will rise today.
then SAL could deduce that the sun will rise tomorrow, the day after tomorrow, and so on.

This may sound silly or dangerous. One may wonder where the deductive process will stop? Won't a rule like that cause an infinite loop? Happily, the answer to the latter is *No, there will be no loop.* The answer to the former is that SAL simply won't start. SAL does no deduction on its own. SAL only replies to queries.

Continuing the example, once SAL has been give the data and a rule, it does nothing. Only when SAL is queried as to the status of the sunrise for a given day will the deductive process begin. So, if one asks SAL: *Will the sunrise tomorrow?*, SAL will use the above fact and rule to *project* the datum: *"(sunrise tomorrow)"*.

To summarize:

SAL has data: SAL manages two different types of data: those that are *always* true, and those that are true at a *particular time*. These are respectively called *factual*, and *temporal* data. Both of these types of data may be provided as input or be the result of projection, i.e. rule firings. The origin of the information is also managed: the origin of the data may be either *user input* or *projection*,

SAL has rules: A *rule* can perform the projection of a new datum. A rule may use any and all facts, *and indirectly* any and all rules in SAL's possession to project new data,

SAL answers queries: SAL will attempt to respond to any data query such as: *Give me the value of toto in the year 2020*, which would be written `(toto 2020)`. It will first simply look for the datum in its internal database. If the value is present, either because it was supplied by the user, or as the result of a previous projection, then it will be returned as the response to the query. If the datum is not available at the moment of the query, then SAL will attempt use rules as needed to project *all* missing data needed to respond to the query. SAL effectively backtracks from the *date* of the query, executing rules as needed, until all missing data is available, and the response to the initial query is available. In the previous example, a query on the status of sunrise for next Wednesday would cause SAL to project sunrise values for every day from today, until next Wednesday. This is due to the form of the rule which can only project sunrise status for a given day based the immediately preceding day.

4.3.2 How to Use the Function `sal:process`, or How to SAL

What are all those arguments to that function `sal:process`? And what does that function do anyway? How do I use it?

SAL processing is invoked by a call to the function `sal:process`.

Indeed, the "process" function in the sal package is the main all singing all dancing entry point to the sal application interface. This function can load data, model definition, rules, and generate reports. We'll have to look at each of these terms before we can get to the discussion of the function arguments.

Data

SAL needs data. When the file `sal.x86f` (x86f extension is used by CMUCL) is loaded, SAL's internal data structures (cf. chapter 9 on page 112) are empty. SAL can do nothing without data. Loading data into SAL means giving it a pointer to a list-of-lists data structure by means of the `data` argument to `process`. This argument can be arbitrarily structured but the leaves must be of the form:
"`(attribute value date)`" or "`(attribute value)`" where *attribute* is a string, *value* is any valid lisp object and *date* is a number (cf. `defdata` function definition, chapter 8.1.6 on page 107 for more information).

SAL creates *sugar function* support for all the attributes that are encountered during the processing of the `data` argument by the function `defdata`. See section 4.3.5 on page 14 for more information on *sugar functions.*

SAL *requires* data. An error will be raised if the `data` argument contains no valid data.

Model

What about attributes that are needed for computation, but for which no values are available at the time when `process` is called? These are defined to be the *model attributes.* These attributes are declared by an evaluation of the function `rf:defmodel`.

The argument `mdl-path` could contain a *pathname object* which points to a file containing calls to `rf:defmodel`. Alternatively, a call to `rf:defmodel` could be evaluated elsewhere. In any case, *sugar function* support will only be created for attributes loaded by means of the `data` argument or defined in an evaluation of a `rf:defmodel` call.

So what does the call to `defmodel` look like? Well here's an example:

```
(rf:defmodel   "Profit"
               "After Tax Earnings"
               "Capital Expenditure"
               "Smoothed Capital Expenditure"
               "Free Cash Flow")
```

In this call we have declared five *model attributes*. Indeed, each of these corresponds to a value derived from other data, and as such could not be created by the initial loading of data via `defdata`.

Reports

SAL's purpose is to answer queries. These are answered in the form of *"comma separated variable"* files called reports. Only attributes specified for reporting will be output to the report files. To declare an attribute for reporting, use the function `defreport`. For examples, check the *Examples* directory for rule files which contain `defreport` calls.

Beyond the declaration of report attributes, a reporting period must be defined. This is done by means of the two arguments: `report-start` and `report-stop`.

The report file will be generated containing values for all reporting attributes over the period $[report-start, report-stop]$. Section 4.3.4 on page 14 contains more information on the format of the report files.

Where will the report be put? Well this depends on the following arguments:

std-out ; default t: If TRUE, then a report will be written to std-out.

file-out ; default t: If TRUE, then a report will be written to a file. This argument and previous one behave independently of each other, i.e. reporting can go to either std-out or file, or both, or neither.

out-file-name : If supplied, file-out report will go to `out-file-name.csv` in directory indicated in `sal-config.lisp`. if not supplied, report will go to `ticker.csv`.

Specific Arguments for Stock Market Analysis

SAL was written to analyze stocks. As such, there are two arguments to `sal:process` which are specific to this particular domain. These are:

`ticker-string` This string should give the ticker name of the stock being analyzed.

`current-year` This argument should contain the current year, as a four digit integer. If not provided, the system time will be used to establish the current year value.

Other Arguments

At this point all the functional arguments have been described. The remaining arguments are used to control SAL's behavior at a technical level. Beware that if TRUE, each of these will have a negative impact on SAL's execution speed.

These are:

verbose default *t*: will control output of SAL's information messages. being done

logging default *nil*: This argument controls SAL's logging. If TRUE then a log will be appended to the file specified in `sal-config.lisp`.

loop-detect default *nil*: If TRUE, SAL will detect looping conditions in rule firings. See Section 4.3.7 on page 22 for more information on rule firing loops.

Now, having looked at all the arguments, we can understand the lisp function definition in which they are specified:

12 ⟨*sal:process-arg-def* 12⟩≡ (43a)

```
(defun process (&key
                ticker-string
                (current-year (wut:current-year))
                (verbose t)
                report-start
                report-stop
                data
                (mdl-path
                 (wut:path-get
                  (cfg-pkg-name-eval "*sm-model-filename-string*")
                  (cfg-pkg-name-eval "*sm-model-path-string*")))
                (std-out t)
                (file-out t)
                out-file-name
                logging
                loop-detect)
```

As you can see, all of the arguments are **keyword** arguments, and are therefore *optional*. However, being optional for **lisp**, doesn't ensure that SAL will be able to function if an incoherent set of arguments is provided.

Here is a summary of the arguments and their meaning:

ticker-string mandatory: The name of the stock to be analyzed.

current-year optional: The 4 digit integer representation of the current year; will default to the current year in system time if not provided.

verbose optional: default *t*: controls SAL's information messages.

report-start optional: if present, will be start date for reporting.

report-stop optional: if present, will be end date for reporting.

data mandatory: an arbitrarily structured tree containing data tuples as leaves.

mdl-path optional: the *pathname object* pointing to the file containing `defmodel` calls. If not present, will default to a value from `sal-config.lisp`.

std-out optional. If true, report will be written to std-out, default is *t*.

file-out : optional. If true, report will be written to a file, default is *t*.

out-file-name : optional string. If present, and if file-out is TRUE, then report will be written to `out-file-name.csv`. If not present, report will be written to `ticker-string.csv`.

logging optional: default *nil*: If TRUE, SAL will append logging data to the log file specified in `sal-config.lisp`.

loop-detect optional: default *nil*: If TRUE, SAL will detect loops in rule firings.

4.3.3 How to Use the model.lisp file

The `model.lisp` is used to inform SAL of attributes for which there may be no data initially available to load into SAL's database. For example, if we have *revenue* and *cost* data, but no *profit* data, then the *profit* attribute should be declared by a call to `rf:defmodel`. This call is most conveniently located in the the file `model.lisp` since `sal:process` will then automatically load it at the appropriate time in the processing sequence.

An example of the declaration of the model attributes *Profit* and *Value* is the following:

```
(rf:defmodel "Profit"
             "Value")
```

The arguments to `rf:defmodel` are strings. The values are case-sensitive, i.e. "Profit" is distinct from "profit". Any number of strings may be supplied as arguments to `rf:defmodel`.

4.3.4 How to Understand Reports and their CSV Files

SAL's reports are so simple to understand that it's almost silly to write this "how-to." Yet, the "how-to" is written, so you may as well read it, too.

A report is a comma-separated file respecting the following standard:

- data is presented as lines of data elements,
- data elements are separated by *commas,* e.g.
  `data,data,data,`
- there are no separators other than *commas,* i.e. no spaces, tabs, etc.
- data elements may be either *strings* or *numbers,*
- all strings are enclosed by double-quotes, e.g.
  `"this string",`
- all numbers are *not enclosed in quotes;* they use the *dot* as the decimal separator, e.g. `,3.14159,`
- a newline character ends each line of data,
- a missing datum is indicated by consecutive *commas,* e.g.
  `"the","next","element","is","missing",,`
- there is a single header line made as per the following with the year numbers running over the reporting period (naturally):
  `"Ticker","ID","Field","yr_2005","yr_2006"`
- the content of the other data lines conforms to the specification given in the header line, e.g.
  `"ibm",4786,"Profit",83000.0,95000.0`

So what about customization of the Report layout? Well, if there is a requirement, everything is possible. For the moment, let's hope that the requirement doesn't appear...

4.3.5 How to Sugar Functions

Sugar functions are at the heart of SAL. They are main application interface given to the user. They are the "L" of "SAL", as in *Language.* They are also used internally in SAL's implementation.

But, they are really only *syntactic sugar,* and so the name!

So what are they, those *sugar functions*? What *syntax* are they sweetening?

As we said, SAL is made to answer queries and SAL needs data to make those answers, and SAL uses rules to deduce new data. We will explain the sugar by means of an example. Let's start with some data:
`(ceo "John Smith")`.

Internally, SAL maintains a database. To query it we would need some kind of SQL-ish statement:

```
> select ceo,
> from data.
 "John Smith"
```

In our sweet syntax, i.e. *sugar function syntax,* we would only have to write:

```
> (ceo)
 "John Smith"
```

So the *sugar function* used here is called `ceo` and it behaves and can be used just like any other lisp function. *Sugar function syntax,* encapsulates database SQL-ish functionality in a user-friendly manner. SAL creates sugar functions dynamically for all data or model attributes that are loaded. The explanation of how this miracle is accomplished can be found in Section 7 on page 61.

Sugar-functions may be evaluated at the lisp command line, or called anywhere in lisp code, to perform both queries and updates to the data set. We will first describe the query calls, then the updating calls [2].

The following examples of sugar-calls have been executed after running the automated reporting:

Sugar Function Queries

We'll remember that data is either always true, i.e. *factual*, or true for a specific date, i.e. *temporal.* It is important to understand that SAL doesn't really know the difference between always true and sometimes true. SAL thinks of dates as either numbers, e.g. 25, -6, 1960, or as the special lisp symbol ***t***. The former correspond to temporal data, and the latter to factual data.

Also, SAL offers some shortcuts and abbreviations for dates:

- no matter what the query, SAL first looks for a corresponding *fact.* So a query about *the speed of light in 1999,* would return the same value, and follow the same search path, as a query for *the speed of light* without reference to a date,
- if no date is given, SAL first looks for a fact, then if none is found, looks for temporal data for the *current-year.*
- if the date d is such that $|d| \leq 100$ then SAL considers it a a *relative date,* i.e. an offset relative to the *current-year.*

Is that confusing? Well, it may well be. The following examples[3] may clarify things...

```
* (ceo)      ; SAL looks for a ceo fact or ceo of current-year.
"John Smith"
```

[2] To use the sugar functions *without* the 'sgr:' prefixes, first run the lisp command: `(use-package "SUGAR")`.

[3] Some blank lines and semicolons have been removed from the lisp output to unclutter the text.

```
NIL
* (ceo t)    ; SAL looks for a ceo fact, only.
"John Smith"
NIL

* (ceo -3)   ; SAL looks for a ceo fact, or the ceo of 3 years ago.
"John Smith"
NIL

* (ceo 2005) ; SAL looks for a ceo fact, or the ceo for the year 2005.
"John Smith"
NIL
```

Are you wondering what all those *nil* values are? If you're not, then maybe you should be! Indeed, *sugar functions* return two values. In the case of queries, the second value is *nil* if the data was loaded or if was *set*, and the value is *t* if it was *projected.* The following examples illustrate this.

```
* (ceo t :project "Fred") ; ceo value is projected to Fred.
"Fred"
T                         ; T indicates that Fred was projected.
*  (ceo)
"Fred"
T                         ; T indicates that Fred was projected.
*  (ceo t :set "Mary")    ; Set is what happens when data is entered
"Mary"                    ; by a call to defdata.
NIL                       ; NIL indicates not projected.
*  (ceo)
"Mary"
NIL                       ; NIL indicates not projected.
```

Indeed, we haven't even finished discussing queries and we are already full into the realm of *updating.* Forgive me, let's hold off on the temptation to explain updating and step back into the world of queries.

Another query form concerns lists of values corresponding to periods of time. Here are some examples which should be self explanatory:

```
* (profit)           ; find the profit for the current year.
85000.0d0
T                    ; indicates a projected value.

* (profit 1999 2004) ; find the profit for [1999, 2004].

(77548.0d0 78396.0d0 75866.0d0 71186.0d0 79131.0d0 86293.0d0)
(T T T T T T)        ; indicates that all values are projected.
```

```
* (profit -1 +1)      ; find the profit for
                      ; [current-year - 1, current-year + 1].
(83000.0d0 85000.0d0 85000.0d0)
(T T T)

* (profit -1 2007)    ; for [current-year - 1, 2007].
(83000.0d0 85000.0d0 85000.0d0)
(T T T)
```

So is that it? Have we finished discussing *sugar function* queries? Well, not quite. There are two more points to address.

- qualification of queries to accept *projected* data or not,
- technical queries to learn what rules may be fired when looking for an attribute's value.

The ":projected?" keyword

The observant reader has already noticed that the second return value of *sugar function* queries is a boolean indicating if the values are *projected* or not. It may be that the user wants to ensure that *projected* values are excluded from the return. This is done by means of the keyword argument **:projected?**. The following examples illustrate various cases of its use:

```
* (revenues 0 3)     ; get the revenues on
                     ; [current-year, current-year +3]
                     ; :projected? is T by default
(95000.0d0 95000.0d0 95000.0d0 110000.0d0) ; 4 values are returned
(NIL T T NIL)        ; 2 values are projected, 2 are not
* (revenues 0 3 :projected? nil)  ; get only NOT PROJECTED revenues
                                  ; on same period
(95000.0d0 NIL NIL 110000.0d0)    ; the projected values are excluded!
(NIL NIL NIL NIL)
* (revenues 0 3 :projected? t)    ; use of t is the same as not
                                  ; using the argument.
(95000.0d0 95000.0d0 95000.0d0 110000.0d0)
(NIL T T NIL)
```

Now, at last, we are almost finished with *sugar function* queries ...

Rule Query

In some special cases, advanced users may wonder how values have been computed, or why values aren't being computed, etc. There is an easy way to see which rules could *potentially* fire when seeking an attribute's value. The following example illustrates how to use **()** or *nil* as an argument to a *sugar function* to obtain this information.

```
* (revenues nil)
(#<Closure Over Function "DEFUN CREATE-SIMPLE-GET-DATA" {58C4DA49}>
 RULE-FUNCS:USE-PREVIOUS)
NIL
```

Wowawowowa! What is that about? If you don't understand the `#<Closure Over Function ...>` then you should probably skip the rest of this section.

So, you're still reading, brave soul, good for you!

When a *sugar function* is called with the single argument *nil*, i.e. the *empty list,* a technical query is launched returning two values:

1. all rules known to generate the attribute, i.e. general rules associated with all attributes, as well as rules specifically associated with the attribute in question,

2. all rules which are specifically associated with the attribute in question.

Both lists are ordered in according to the way that the rules would fire.

In the previous example, we see that there are two general rules associated with the attribute **revenues**:

1. `#<Closure Over Function "DEFUN CREATE-SIMPLE-GET-DATA" {58C4DA49}>`

2. `RULE-FUNCS:USE-PREVIOUS`

We see that *nil* as second return value indicates that there are no rules specifically associated with the attribute **revenues**. Thus we can deduce that the two rules mentioned are general rules associated with all attributes. As an aside, one can see that the first general rule is a *closure*, and not simply a function name. In fact, this rule is internally defined and used by SAL to query the database. The other rules, those which are indicated by their names, are user defined.

Here is another, more complete example:

```
* (profit ())
(#<Closure Over Function "DEFUN CREATE-SIMPLE-GET-DATA" {58C54871}>
 RULE-FUNCS:SIMPLE-PROFIT RULE-FUNCS:USE-PREVIOUS)
(RULE-FUNCS:SIMPLE-PROFIT)
```

This time we see that there are three rules indicated in the first return value. This is the union of general rules and rules specifically associated with **profit**:

1. `#<Closure Over Function "DEFUN CREATE-SIMPLE-GET-DATA" {58C4DA49}>`

2. `RULE-FUNCS:SIMPLE-PROFIT`

3. `RULE-FUNCS:USE-PREVIOUS`

We note that the additional rule, `SIMPLE-PROFIT` appears in the middle of the list. Why would that be? Because the list is returned in the *order of* *precedence,* i.e. the order in which the rules will be fired when seeking a value of **profit**.

The second return value is a list which contains the single rule which is specifically assigned to **profit**: `RULE-FUNCS:SIMPLE-PROFIT`

At this point, dear reader, you can say that you know all there is to know about *sugar function* queries!

Sugar Function Updates

After reading up to this point, you should feel comfortable with the use of *sugar functions* as a convenient means to query the database. That's fine, but what about updating? You may need to load and/or update *projected* data and/or new data which is not deduced. This is the second usage for *sugar functions.*

As we have previous mentioned, data enter SAL in two forms:

projected data: data which the user has computed, or deduced, based on rules and/or other data and which is declared by a *sugar function :project* call.

set data: data which is entered via direct loading through **defdata** or **process**, or which is declared as *set* by a *sugar function :set* call.

Both of these forms of data may be expressed by *sugar functions* of the form $(<\textit{attribute}><\textit{date}><\textit{keyword}><\textit{value}>)$. The following examples illustrate this:

```
* (industry)
"COMPUTER"
NIL
* (industry t :set "garbage") ; set industry's value to "garbage",
"garbage"                     ; date 't' indicates factual data.
NIL
* (industry)                  ; query shows "garbage" NOT PROJECTED.
"garbage"
NIL

* (profit)                 ; query profit for current year
85000.0d0
T                          ; it is a PROJECTED value
* (profit 0 :set 99)       ; SET profit for current year
99                         ; note date argument is NOT OPTIONAL with
                           ; :set or :project keywords!
NIL                        ; it is a NOT PROJECTED value
* (profit)
99
NIL
* (profit 0 :project 88)   ; project the value for current year
88
T                          ; T indicates a PROJECTED value
```

```
* (profit)                ; query confirms!
88
T
```

That was easy, wasn't it!? But by now the reader should expect some spice in each section, and here it comes. *sugar functions* can also be used for other, more technical forms of updating. The following forms may also be used:

(<attribute> :model <indicator>) : This form is used to declare a *model* attribute and to indicate its membership to the list of *reported* attributes or not,

(<attribute> :rule <name> :prec <precedence>) : This form associates a *rule function* with an attribute.

The first of the above forms is used to declare a *model* attribute and to indicate if it should be reported or not (cf. section 4.3.3 on page 13 for more information on *model attributes*).

Attention, the values of the indicator are subtle:

t : indicates a MODEL attribute that is to be REPORTED,

true value other than *t* : indicates a MODEL attribute is NOT to be reported,

***nil* value** : indicates NOT a MODEL attribute.

The following examples should make this clear:

```
* (profit :model t)    ; declare profit both MODEL and REPORTED
"Profit"               ; the string name of proift is "Profit"
T

* (profit :model 1)    ; declare profit MODEL and NOT reported
"Profit"
T

* (profit :model nil)   ; declare profit NOT model and NOT reported
"Profit"
NIL
```

The second form is used to associate a *rule function* with an attribute at a specific *precedence* value which must be > 0 (cf. section 4.3.7 on page 22 for more information on *rule functions).*

Below we define a (boring) *rule function* and associate it with **profit** at a *precedence* value of 20:

```
* (defun my-rule (&rest args)
    "this is an empty-rule that will always fail."
```

```
    (declare (ignore args))   ; to prevent lisp warnings
    (values ()()))   ; definition of the rule function
MY-RULE
* (profit :rule #'my-rule :prec 20) ; association to "profit"
                                    ; with precedence = 20
(#<Interpreted Function MY-RULE {58536831}> . 20)
T
* (profit ())           ; query the rules for "profit"

(#<Closure Over Function "DEFUN CREATE-SIMPLE-GET-DATA" {58C54871}>
 RULE-FUNCS:SIMPLE-PROFIT #<Interpreted Function MY-RULE {58536831}>
 RULE-FUNCS:USE-PREVIOUS)
(RULE-FUNCS:SIMPLE-PROFIT #<Interpreted Function MY-RULE {58536831}>)

; my-rule appears as the only specific rule association for "profit"
```

4.3.6 Special Sugar Functions

We have seen that *sugar function* are automatically created for all attributes as they are introduced to SAL. In addition, there are few *special sugar functions* created by SAL itself. These should be used with *extreme* caution if they are called in update forms! The following are the *special sugar functions*:

general-rules : This sugar function will process the general rules known to SAL.

model-attributes : This sugar function will process the list of model-attributes.

Here are some examples of their use in *query* forms:

```
* (use-package "SGR")
T

* (general-rules)
(#<Closure Over Function "DEFUN CREATE-SIMPLE-GET-DATA" {58C3BB01}> . 0)
((RULE-FUNCS:USE-PREVIOUS . 100))

* (model-attributes)
("Loopy" "Profit" "After Tax Earnings" "Capital Expenditure"
 "Smoothed Capital Expenditure" "Free Cash Flow" "Employees"
 "Manufactured Goop" "Manufactured Toto")
T
```

Summary of Sugar Function Forms

The following is the exhaustive list of *sugar function* forms:

(< *attribute* >) : query for attribute's fact or current year value,

($<$ *attribute* $>$ **t**) : query for attribute's fact value,

($<$ *attribute* $>$ $<$ *date* $>$) : query for attribute's fact or date value where *date* $<$ 100 indicates a value relative to the current year, and the date value t indicates a fact value only,

($<$ *attribute* $>$ $<$ *start* $>$ $<$ *end* $>$) : query for attribute's fact or date value for each date on $[start, end]$,

($<$ *attribute* $>$ $<$ *date* $>$ **:projected?** $<$ *bool* $>$) : query for attribute's fact or date value accepting *projected* values if the boolean is true or refusing if false,

($<$ *attribute* $>$ $<$ *start* $>$ $<$ *end* $>$ **:projected?** $<$ *bool* $>$) : as previous but for fact or date value for each date on $[start, end]$,

($<$ *attribute* $>$ ()) : query for rule functions associated with attribute,

($<$ *attribute* $>$ $<$ *date* $>$ **:set** $<$ *value* $>$) : set attribute's value for date,

($<$ *attribute* $>$ $<$ *date* $>$ **:project** $<$ *value* $>$) : project attribute's value for date,

($<$ *attribute* $>$ **:model** $<$ *indicator* $>$) : declare attribute as reported and model if indicator is a *t,* declare as model but not for reporting if indicator is a *true value* $\neq t$, declare not a model attribute if indicator is *false,*

($<$ *attribute* $>$ **:rule** $<$ $rule\ function$ $>$ **:prec** *number*) : associate rule function with attribute at precedence number - note that precedence must be > 0.

4.3.7 How to Write a Simple (single-attribute) Rule

This How-To explains the procedure used to create a rule which will handle queries to a single attribute.

What is a Rule?

A rule is simply a lisp function which meets the following requirements:

1. It is called with 2 arguments:
   (a) an attribute as a string,
   (b) a date which could be either t or a number.
2. It returns two values:
   (a) either the value of the attribute for that year, or *nil* if not available,
   (b) t if the value is *projected* or *nil* otherwise.

The rule function may also have side effects, such as writing the value of the attribute for the date into the internal database, but this is not required. It may seem surprising to see that a rule is not required to update the internal database with the value it computes. In fact, this lack of constraint allows the system's own lookup mechanism to be written as rules. Of course, it would be much more efficient if, as a side effect, the rule were to write any computed data to the database (with a `:project` or `:set` call, for example). If not, the same computation may be performed over and over again during processing.

So in summary, a rule can be any lisp function that meets the requirements listed above. Here are some simple and even silly examples:

```
(defun does-nothing (att date)
  "This rules does nothing and failes."
  (declare (ignore att date))   ; to avoid warnings
  (values ()()))

(defun always-10 (&rest un-used)
  "This rule will return the value 10, not-projected.
  It will not update the database"
  (declare (ignore un-used))   ; to avoid warnings
  (values 10 ()))

(defun complicated-rule (att date)
  "If the date is a number, it is the value returned, otherwise
  use 0. In both cases, it is a Projected value."
  (declare (ignore att))   ; to avoid warnings
  (values (if (numberp date) date 0) t))
```

Each of these silly rules complies with the requirements and could therefore be used by the analysis engine. After a short digression on the analysis engine in the following section, we will then discuss the writing of rules! Please be patient . . .

When are rules called?

The analysis engine is brought into play each time a query is made by means of a sugar-function. For example, a call such as:

```
* (profit 2050)
```

will set the analysis engine into operation, searching for a value for the attribute `profit` for the year 2050. The engine will search for the value by first looking in the internal database for the corresponding value. This could be either a factual value which is true for all times, or a datum which is true for the specific date requested, in this case the year 2050.

If neither of these values is available, the analysis engine will then begin applying all rules which have been registered as applicable to the attribute `profit`, including rules which apply to all attributes (generic rules). These rules will be applied in order of increasing precedence until either a value is discovered, or all the rules fail. In both cases the query is considered as terminated, i.e. no error is generated.

Now that sounds simple doesn't it? Well it is simple, but what happens if a rule needs data about an `(attribute, date)` pair to produce a value to answer the query? In the example, suppose that there were a rule that said, *"Profit for a year is the difference between revenues and costs for that year."* When this rule fires on the attribute `profit`, for the year `2050`, it will generate two new queries: `(revenues 2050)` and `(costs 2050)`.

These two queries will again set the analysis engine into action, looking first in the database, then applying rules to try to find values if needed. The only limit to the depth of this recursive search is that imposed by the lisp environment.

If it still sounds simple, then you are doing well and keeping up.

So, what can go wrong? Well, suppose that a sequence of rule firings is circular, i.e. there is a loop. Continuing our example, suppose we had an additional rule: *"Revenues for a year are the difference between profit and costs."* Now we have a potential loop which could occur as follows:

1. query `(profit 2050)`
2. profit rule fires generating the query `(revenues 2050)`
3. revenues rule fires generating the query `(profit 2050)`
4. profit rule fires generating the query `(revenues 2050)`
5. etc.

This is a loop and it is a bad thing since it will simply continue until there is no more memory available, then in the worst case, crash lisp. Luckily, SAL can detect such loops if the argument to `process` *loop-detect* is set to TRUE (cf. section 4.3.2 on page 10 for more information.)

This explanation should have made it clear that rules are fired when data is needed, but is unavailable in the database.

You should now be ready and perhaps even anxious to learn to write a rule.

Where do Rules Come From?

Well by now the reader should have guess that the `sal-config.lisp` is the source of all user configuration parameters. Let's take a few moments to look at how SAL uses that config data to find rules.

The basic idea is that a special attribute called "industry" must be available to SAL. This is used for indirection to find which sets of rules should be applied. These sets are defined by the configuration parameters:

- `*sm-industry-rulefile-string-alist*` is used to associate industries with rule files,

- `*sm-model-path-string*` contains the path to the rule files.

The special alist key *t* points to the "default-rules" which are taken to apply to all industries.

See section 11.1.2 on page 151 for more details on these and other configuration parameters.

Writing a Simple Rule

Now let's look at the rule writing process. We've seen that rules are used to fill in missing information, or to *project* new data based on other data. Let's look back at the previous example: *"Profit for a year is the difference between revenues and costs for that year."*

In lisp, this is simply expressed by the following function:

```
(defun profit-rule (att date)
  "This version returns the profit value as projected, but does not
  save it to the database."
  (declare (ignore att))
  (values (- (revenues date) (costs date)) ; first value is the profit
          t))                               ; second indicates Projected.
```

This simple rule function satisfies the requirements and and does the job. There are several points to recognize in this rule:

- the *sugar functions* **revenues** and **costs** were used to query the database for their values for the *year*.
- we assumed that the values for **revenues** and **costs** will always be available! Imagine what would happen if either of those returned () ...
- At this point the function **profit-rule** is known to lisp, but not to SAL.

We must still associate it with the attribute profit. There are two ways of doing this:

- by means of *sugar function* rule association,
- by restructuring the definition in a call to `defrule`.

If the rule has already been defined as a lisp function, then the easiest solution would be to use *sugar function* rule association. The following example associates the function **profit-rule** with the attribute *profit* at *precedence* value 20:

```
(profit :rule #'profit-rule :prec 20)
```

This works fine. No problem. But it requires two steps, thus leaving plenty room for error. Would it be nice to be able to perform the definition of the rule and the association to attributes, yes a rule may be associated with *several attributes,* in a single statement? Well that is exactly the purpose of the SAL function `defrule` which is available in the "RULE-FUNCS" package. Here's the profit rule reformulated into a `defrule` call, with protection for missing revenues or costs data, exclusion of the case that the rule was called on the date *t* (meaning on the *always true* value of profit which is obviously erroneous), and with an embedded *sugar function* update call to write the computed value to the internal database:

```
(in-package "RULE-FUNCS")

(defrule simple-profit ("Profit") 20 (un-used yr)
  " profit = revenue - costs;
  but is nil if either of the components is missing."
  (declare (ignore un-used))
  (if (not (numberp yr)) (values ()())   ; we exclude facts
    (let* ((revenue (revenues yr))
           (costs   (costs yr))
           (val (and revenue costs (-  revenue costs))))
      (if val (profit yr :project val)
        (values ()())))))
```

So what's all that about? First, what is the form of a `defrule` call?

```
(defrule <name>
  <attribute-list>
  <precedence>
  <formal-parameters>
  <body>)
```

What are the arguments to `defrule`?

<**name**> : this is simply the symbol-name as in the lisp function definition,

<**attribute-list**> : a list of *strings* which correspond exactly, including case, to the attributes to which the rule is to be associated,

<**precedence**> : a number > 0, lower precedence values indicate earlier execution than higher values,

<**formal-parameters**> : there are two parameters, but &rest could be used to group them into a single list,

<**body**> : the body as per any lisp function definition.

Now we can understand the `defule` call:

- We first move to the proper package: RULE-FUNCS.

- The first line says that a rule named *simple-profit* is to be associated with the attribute "$Profit$" at precedence 20; the formal parameters are *un-used* and *yr*,
- we next tell lisp to *ignore* the formal *un-used* which won't be used in the function,
- continuing, if *yr* is not a number, then the rule was called with $yr = t$, this case is not handled so we return *nil nil*,
- using *sugar* calls, we try to obtain values for *revenues* and *costs* corresponding to the *yr*,
- the result *val* is the difference, but only if both *revenues* and *costs* are not *nil*,
- if a result *val* was obtained, return the result of assigning it to $profit$ for the *yr* as projected,
- if not, return *nil*;*nil*.

What have we learned so far? That `defrule` is the way to go for rule creation and association. In a single call, a rule is defined and associated with proper precedence to the selected attributes. Furthermore, the form of the `defrule` call allows for many levels of abstraction to be applied. For example, suppose that we would like to use a specific key value to indicate that a *fact* was projected in specific circumstances. Also, we would like this to work for several attributes: "CEO", "DHR", "CTO". Here's the code:

```
(in-package "RULE-FUNCS")

(defrule brother-in-law ("CEO" "DHR" "CTO") 20 (at yr)
  "We assume the lack of data means that the
  brother-in-law is the person doing the job.
  "
  (if (numberp yr)     ; we exclude timely data
      (values ()())
    (apply-attribute-sugar at yr :project "the-brother-in-law")))
```

Here is the full trace from CMUCL:

```
* (in-package "RULE-FUNCS")
#<The RULE-FUNCS package, 23/40 internal, 13/21 external>

* (defrule brother-in-law ("CEO" "DHR" "CTO") 20 (at yr)
  "We assume the lack of data means that the
  brother-in-law is the person doing the job.
  "
  (if (numberp yr)     ; we exclude timely data
```

```
      (values ()())
    (apply-attribute-sugar at yr :project "the-brother-in-law")))

#<Interpreted Function (LAMBDA (AT YR)
                         "We assume the lack of data means that the
  brother-in-law is the person doing the job.
  "
                         (IF # # #))
  {5854B469}>

* (sgr:ceo ())
(#<Closure Over Function "DEFUN CREATE-SIMPLE-GET-DATA" {58C4B421}>
 BROTHER-IN-LAW USE-PREVIOUS)
(BROTHER-IN-LAW)

* (sgr:ceo)
"the-brother-in-law"
T

* (sgr:dhr 2006)

"the-brother-in-law"
T
```

That should be clear and almost ends our exploration of the definition of simple rules. Indeed, the attentive reader will have noticed that the last example slipped in a little indirection on the attribute name by means of the function `apply-attribute-sugar` in the **"SUGAR"** package. Indeed, this function and its friends, `attribute-sugar-function`, and `att-name-2-sugar-func` pave the way to writing *generic* rules.

Summary of Simple Rules

Simple rules are, well, *simple.* They are simply lisp functions that are associated to attributes and called when the attribute's value is needed but not available in the database.

Rule functions have two formal arguments:

attribute : which will bind to a string that exactly matches the attribute's name,

date : which will bind to a number or t.

Rule functions can be associated to attributes in two manners:

- by $sugarfunction$: `(sgr:profit :rule #'a-profit-rule :prec 30)`
- by means of `defrule`: `(defrule a-profit-rule ("Profit") 30 (at yr) ...)`.

The same rule function can be associated to several attributes, using either of the above methods.

You now know all there is to know about simple rules!

4.3.8 How to Write a Generic (multi-attribute) Rule

At this point the reader knows how SAL looks for data, and how rules are fired to provide that data. In this section we will explore the use of *generic rules* that work on multiple attributes.

There are two types of generic association: explicit association with a set of attributes and association with *all* attributes.

In the previous section we saw that *rule functions* can be associated to attributes either by *sugar calls* or by means of `defrule`. Both of these methods can be used to produce multi-attribute associations. However, only `defrule` can perform association to *all* attributes. We will look at this first by means of the example of the standard rule called `use-previous`. This rule says: *For any timely attribute, the value for the date d is simply the value for the date* $d-1$.

Here is the implementation:

```
(defrule use-previous (t) 100 (at yr)
  "This rule says, if the previous year is not less
than the rist year of data, then use the previous year's
attribute value for this year.
Careful, a year could be 't'.
"
  (if (not (numberp yr)) (values () ())  ; reject facts
    (let* ((previous (1- yr))
           (min-year *first-year-of-data*)
           (val (and (>= previous min-year)
                     (apply-attribute-sugar at previous))))
      (if val (apply-attribute-sugar at yr :project val)
        (values () ())))))
```

Having read and understood all the previous How-To's, this rule should read like "see spot run." Here's how it works:

- the rule's name is "use-previous",
- it associates with the list (t), meaning *all* attributes,
- it's precedence is 100, which means that it will only fire if all lower precedence rules have failed to find a value,
- the first line of code filters out any calls on facts, i.e. returns failure values, since this rule only applies to timely data,
- *previous* is bound to $yr-1$,
- *min-year* is bound to the global value of the minimum date,

- the new *val* is bound to *nil* if the previous year is too early or if *there is no value for the attribute for the previous year.* Note that the `apply-attribute-sugar` call could provoke a recursive firing of the *use-previous* rule, and this is proper behavior!

- finally, if the new value is not *nil*, then it is *projected* as the value for the attribute for the year. The value of the call to `apply-attribute-sugar` is the value returned by the rule,

- if the new value is *nil*, then return the failure values.

This should be clear to all. It is worth noting that the value of *at* is never explicitly examined. That is indeed the semantics of a *generic rule*.

A generic rule may not be designed for *all* attributes, but a limited set. This is accomplished by listing them explicitly in the second argument to `defrule`, or by making a specific *sugar* call to associate the rule function with each attribute.

Funnily, generic rules are even *simpler* that simple rules!

4.4 Requirements

SAL's requirements are both few and straightforward. They are detailed in the following sections:

4.4.1 Functional Requirements

1. The application shall provide facilities to extrapolate data based on inputs and computation functions,

2. Extrapolated (*projected*) data shall be segregated from input data,

3. Input data representation shall permit the expression of time dependent and time independent data,

4. Bulk and individual data element loading shall be provided,

5. User interface shall provide functional style access to attribute values, e.g. **(profit 2001)**. This is only an example, final representation to be defined in design,

6. Reporting shall provide csv output with columns as:
   `"Ticker","ID","Field","yr_2005","yr_2006"`,

7. Program shall be executable independently for each family of attributes, i.e. data corresponding to different *tickers* shall not interact,

8. The application shall permit parallel execution on different data sets,

9. Projection facilities for new values for data shall be provided.

4.4.2 Standard Software Engineering Requirements

1. The application shall comply with the idioms of the programming language used in the implementation: Lisp,
2. System installation and upgrading shall be feasible without detailed understanding of the programs,
3. Local user configuration shall be available to the user,
4. The application shall be fully documented such that its maintenance does not depend on the presence of the original development team,
5. The application shall be based on state-of-the-art design principles avoiding ad hoc solutions whenever possible. In particular, automated test harnesses shall be provided.

4.5 Architecture

4.5.1 System Philosophy

The idea behind SAL is that of a backward chaining expert system shell. Data is stored in memory and queries are handled by SQL-like look-up, calling on rules to project missing data. Underlying support for recursion is provided by Lisp, and maintained throughout the application.

4.5.2 System Structure

The SAL application comprises:

Internal Database : This is the core of the application, providing the underlying data representation and basic create, read and update functionality. The internal database provides the lowest level of functionality and is intended for internal use by the system itself. It is not intended to provide end-user, or API level functionality.

Sugar Function : This layer relies on the previous one for data and provides the user level querying and updating facility described in How-To's. It relies on *rule functions* to perform data extrapolation, i.e. projection. Much of SAL's internal code is written in *sugar functions* which are built using the support provided by this module.

Rule Function : This small module provides support for the creation and manipulation of the rule functions.

Top Level : The applicative interface sits here. This module performs the loading and unloading of data and provides the processing and reporting functionalities.

Chapter 5

Makefiles

This is the description of the **Makefiles** which are used to build and install SAL.

There are two makefiles associated with the construction of SAL. The first, `Makefile.0`, is used to build the second: `Makefile`. It is the second `Makefile` which is used in the building, installing and testing of SAL. The second `Makefile` is also used to build the SAL distribution archive.

The `Makefile` provided in the distribution of SAL is in fact `Makefile.0` renamed to `Makefile` so as to enable the call to **make** with no arguments.

To build, install and test SAL, once the user configuration (cf. section 3 on page 3) has been set-up, simply run the commands:

```
$ make
$ make
$ make examples # the example files are needed for the tests.
$ make install wut-test ids-test sugar-test sal-test
```

The first call to **make** uses the `Makefile.0` to build the full `Makefile`. The second call builds SAL and its documentation, the third installs the example files which are needed to run the test suites, and the final call with the **install** target, installs SAL and runs the unit tests (not mandatory).

To build a full distribution archive, execute the following command:

```
$ make dist
```

This will create a file **sal.tgz** in the current directory containing the full archive, including any customizations that may have been made.

5.1 Makefile.0

Makefile.0 is the makefile used to build the final Makefile which is used to build SAL. The content of Makefile.0 is described here since it is provided as the *only* Makefile in the SAL distribution. This allows for a straightforward make process by anyone, without the need for the user to perform any *noweb tangle* or *weave* commands.

Makefile.0 first defines the tangle and the `emacs-commands` (see below) it will use and the files which will be the prerequisites.

33a ⟨*Makefile.0* 33a⟩≡ 33b ▷

```
# commands
TANGLE=notangle

⟨emacs-commands 40c⟩

#files:
NW_FILES= user-config.nw makefile.nw
```

Then, the target **all** is defined as depending on the noweb files defined above. The actions required to make the main SAL Makefile are:

1. first backup the current copy of Makefile to Makefile.0,
2. then *tangle* out the real Makefile (note the use of the **-t8** option which converts eight spaces into a tab character, thus allowing the make utility to find commands for building the targets). For those unfamiliar with gnu make, the `$^` symbol replaces the prerequisites, which in this case are the noweb files, and the `$@` symbol replaces the current target which is `Makefile` at this point. Also the use of `.PHONY` ensures that the target `Makefile` will always be built, even if a file exists of the same name which is current with respect to the prerequisites,
3. and finally use *emacs* to clean up the newlines which are not properly handled by noweb, it seems.

33b ⟨*Makefile.0* 33a⟩+≡ ◁ 33a

```
# targets:
.PHONY: Makefile

all: Makefile

Makefile: $(NW_FILES)
        -cp $@ $@.0
        $(TANGLE) -t8 -R$@ $^ > $@
        $(EMACS_CMD) $@ $(EMACS_OPTS)

⟨make-Makefile.0 33c⟩
```

If for some reason the file makefile.nw is updated then Makefile.0 will need to be tangled out of the makefile.nw file. The following make target will do that. It is included in both makefiles.

33c ⟨*make-Makefile.0* 33c⟩≡ (33b 41)

```
Makefile.0: $(NW_FILES)
        $(TANGLE) -t8 -R$@ $^ > $@
        $(EMACS_CMD) $@ $(EMACS_OPTS)
```

If it becomes necessary to manually extract `Makefile.0` as `Makefile` from makefile.nw, this is done by the following command:

```
$ notangle -t8 -RMakefile.0 makefile.nw > Makefile
```

After installation, a call to make with the target **clean** will remove all the generated files, but *only in the current directory*, tangle `Makefile.0` out of makefile.nw, and copy it to `Makefile`.

5.2 The SAL Makefile

The SAL `Makefile`, like any makefile is composed of several parts: variable definitions, high level targets and detailed or *workhorse* targets.

5.2.1 Variable Definitions

The variables used in the Makefile ensure that the command specifications embedded in the targets can remain unchanged, even if programs, paths, etc. are updated in the system or the build platform. Further unification and preservation against change is achieved by the shared use of noweb code chunks to isolate user configuration to the single file *user-config.nw* (cf. chapter 3 on page 3).

34 ⟨*makefile variables* 34⟩≡ (41)

```
# user configuration parameters:
SAL_BIN_FILE=⟨bin-file-path 3c⟩sal.x86f
SAL_TEST_DATA_FILE=⟨model-rule-path 3d⟩data.lisp
SAL_CONFIG_PATH=⟨configfile-path 3a⟩
SAL_MAKE_INSTALL=sal-build:make-install "$(SAL_CONFIG_PATH)"
SAL_DIST_ARCHIVE=sal.tgz

# commands:
WEAVE=noweave
WEAVE_OPTS=-autodefs lisp -delay -index
WEAVE_HTML_OPTS=-filter l2h -html
HTML_TOC=htmltoc -12345
TANGLE=notangle
LATEX=latex
DVIPDF=dvipdf
LISP_ENVT=export FINDUSES_LISP=1
LISP=⟨lisp-path 4d⟩

#files:
NW_FILES=opening.nw \
intro.nw \
user-config.nw \
doc.nw \
makefile.nw \
```

```
sal.nw \
sugar.nw \
rule-funcs.nw \
internal-data-structure.nw \
utilities.nw \
sal-config.nw \
sal-build.nw \
closing.nw

CONFIG_FILE=sal-config.lisp

LISP_FILES=sal-build.lisp \
internal-data-structure.lisp \
$(CONFIG_FILE) \
sal.lisp \
sugar.lisp \
rule-funcs.lisp \
utilities.lisp

SAL_BUILD_FILE=sal-build.lisp

EXAMPLES_ARCHIVE=examples.tgz

DIST_DOC=sal.dvi README
DOC_FILES=$(DIST_DOC) sal.pdf sal.html

DIST_FILES=$(NW_FILES) $(EXAMPLES_ARCHIVE) $(DIST_DOC)
```

5.2.2 High Level Targets

A Makefile should usually specify high level or *abstract* targets. These are often things like **all** (the first and therefore the default target) or **install**. SAL's `Makefile` specifies quite a few:

The following targets may be of particular interest:

install: The **install** target has for prerequisite the abstract target **code**. The command to build it is relatively straightforward. The lisp program shall load the value of the variable in the **SAL_BUILD_FILE**, then evaluate the lisp expression which after variable substitution produces **'(progn (sal-build:make-install <path-to-config>) (quit)'**. In other words, load the file sal-build.lisp and execute make-install.

dist: This target has 3 abstract prerequisites and is used in building a distribution archive.

clean: This *PHONY* target and has no prerequisites. As a *PHONY,* there will be no checking to see if it is up to date, which makes sense since we want to be able to execute the clean commands under all circumstances. It simply deletes all the products and by-products of building the other targets as well as `Makefile.0`. It then rebuilds `Makefile.0` and copies it back to `Makefile`. The first call to make after a **make clean**, builds the full `Makefile`. The second call can build genuine targets.

36 ⟨*abstract-targets* 36⟩≡ (41)

```
all: code doc

install: code
        $(LISP) -load $(SAL_BUILD_FILE) \
-eval '(progn ($(SAL_MAKE_INSTALL)) (quit))'
        cp $(CONFIG_FILE) $(SAL_CONFIG_PATH)

code: $(LISP_FILES)

doc: $(DOC_FILES)

dist: dist-files dist-build

.PHONY: clean
clean:
        -rm -fv *.aux *.dvi *.html *.log *.pdf *.tex *.toc  $(LISP_FILES)
        -rm Makefile.0
        $(MAKE) Makefile.0
        cp  Makefile.0 Makefile
```

5.2.3 Workhorse Targets

A Makefile must have some targets that actually do the building. These are the workhorses of the `Makefile`. Before looking at these in detail, it is necessary that we understand the meaning of some special symbols in the gnu make syntax. These are more fully documented in the *GNU Make Manual*.

% in target or prerequisite spec.: The `%` wild-card is similar to `*` on the Linux command line. It will match any name. Once it has been matched, that same name will replace all instances of `%` in either targets or prerequisites. So, a rule starting like:

```
%.x86f: %.lisp  %-build.lisp
```

would mean that a target such as **toto.x86f** will depend on the prerequisites `toto.lisp toto-build.lisp`.

$* in the command spec.: This pattern matches the expression that has previously been bound to a `%` symbol. In the previous example, this would bind to `toto`. This pattern can be used anywhere in the command line.

$@ in the command spec.: The `$@` pattern matches the current build target. In the previous example, `$@` would match **toto.x86f**. This pattern can be used anywhere in the command line.

$< in the command spec.: The `$<` pattern matches the 1st of the current build prerequisites. In the previous example, `$<` would match **toto.lisp**. This pattern can be used anywhere in the command line. It will only match the single first name in the prerequisites.

$^ in the command spec.: The `$^` pattern matches the entire list of current build prerequisites. In the previous example, `$^` would match `toto.lisp toto-build.lisp`. This pattern can be used anywhere in the command line. It should be noted that this pattern will match more than one file name, as is the case in the rule for the target `%.tex`.

The following targets may be of particular interest:

%.pdf: This very simple target says: "To make a `pdf` file, the corresponding `dvi` must be up to date, then to build it run the DVIPDF command on the `dvi` file." A similar target exists for `dvi` targets.

37 ⟨*detailed-targets* 37⟩≡ (41) 38a ▷

```
%.pdf: %.dvi
        $(DVIPDF) $<
```

%.dvi: Again a very simple target. Note that the LaTeXcommand is run three times so as to ensure that all cross references are resolved.

38a ⟨*detailed-targets* 37⟩+≡ (41) ◁37 38b▷
```
%.dvi: %.tex
	$(LATEX) $<; $(LATEX) $<; $(LATEX) $<
```

%.tex: This target depends on the noweb files being up to date. Building it is done in two parts. First the **LISP_ENVT** is created. Then, following a semi-colon which ensures that the next command will be executed in the same shell, we *weave* the prerequisite noweb files into the **tex** target.

38b ⟨*detailed-targets* 37⟩+≡ (41) ◁38a 38c▷
```
%.tex: $(NW_FILES)
	$(LISP_ENVT); $(WEAVE) $(WEAVE_OPTS) $^ > $@
```

%.html: This is similar to the previous with the additional step that the output from *weaving* is piped through an html table-of-contents filter before being directed to the **html** target.

38c ⟨*detailed-targets* 37⟩+≡ (41) ◁38b 38d▷
```
%.html: $(NW_FILES)
	$(LISP_ENVT); \
$(WEAVE) $(WEAVE_OPTS) $(WEAVE_HTML_OPTS) $^ | $(HTML_TOC) > $@
```

LISP FILES: This is again similar to the previous targets. The main difference is that we are now *tangling* i.e. building code. The argument to the `-R` option of `notangle` indicates which code chunk should be taken as the root to be used to build the target. In this rule, the `$@` argument is the name of the lisp file target. This is the convention that has been used in the noweb sources: "root chunks have the name of the target file." The second line of commands is a call to emacs in batch mode which fixes the newlines generated by noweb.

38d ⟨*detailed-targets* 37⟩+≡ (41) ◁38c 39a▷
```
$(LISP_FILES): $(NW_FILES)
	$(TANGLE) -R$@ $^ > $@
	$(EMACS_CMD) $@ $(EMACS_OPTS)
```

%-test: This is a different type of target than the previous ones in that it doesn't build anything. Its prerequisites are the lisp files. The commands simply load `sal` and the test data into lisp, then run the corresponding test harness by a call to the `test` function in `utilities.lisp`, and exit the lisp environment.

39a ⟨*detailed-targets* 37⟩+≡ (41) ◁38d 39b▷

```
%-test: $(LISP_FILES)
	$(LISP) -load $(SAL_BIN_FILE) \
-load $(SAL_TEST_DATA_FILE) \
-eval '(progn (wut:test "$*") (quit))'
```

%-test-no-quit: This is slight variation of the previous. The only difference is that the lisp environment is not closed after running the test.

39b ⟨*detailed-targets* 37⟩+≡ (41) ◁39a 39c▷

```
%-test-no-quit: $(LISP_FILES)
	$(LISP) -load $(SAL_BIN_FILE) \
-load $(SAL_TEST_DATA_FILE) \
-eval '(wut:test "$*")'
```

examples: This target unpacks example rule, model, and data files to the Examples subdirectory of the binaries directory.

39c ⟨*detailed-targets* 37⟩+≡ (41) ◁39b 39d▷

```
examples: $(EXAMPLES_ARCHIVE)
	tar xvfz $< -C ⟨bin-file-path 3c⟩
```

dist-files: This is the target that will make a clean build of all the files in preparation for the creation of a distribution archive.

39d ⟨*detailed-targets* 37⟩+≡ (41) ◁39c 40a▷

```
dist-files:
	-$(MAKE) clean
	$(MAKE)
	$(MAKE)
```

dist-build: This will create the distribution archive. First is checks that the **DIST_FILES** and **EXAMPLES_ARCHIVE** are available. It then creates a temporary directory, copies the files there, makes a gzip archive of them, and moves that archive back to the current directory. Finally, it removes the temporary directory.

40a ⟨*detailed-targets* 37⟩+≡ (41) ◃39d 40b▹

```
dist-build: $(DIST_FILES) $(EXAMPLES_ARCHIVE)
        -mkdir /tmp/Build
        -rm -vf /tmp/Build/*
        cp Makefile.0 /tmp/Build/Makefile
        cp $^ /tmp/Build
        cd /tmp/Build; tar -cvzf $(SAL_DIST_ARCHIVE) *
        mv /tmp/Build/$(SAL_DIST_ARCHIVE) .
        -rm -rfv /tmp/Build
```

dist-clean: This is used to clean up the distribution archive in the event that it was incorrectly built.

40b ⟨*detailed-targets* 37⟩+≡ (41) ◃40a

```
.PHONY: dist-clean
dist-clean:
        -rm -vf /tmp/Build/*
        -rm -fv $(SAL_DIST_ARCHIVE)
```

5.3 Makefile Utilities

The following *emacs* variables assemble to make an `emacs` command that replaces the noweb inserted `<CR>` by a proper Linux newline. Although the difference between newline and `<CR>` is not generally significant, there are particular circumstances, such as in file names, where it is important.

40c ⟨*emacs-commands* 40c⟩≡ (33a 41)

```
# emacs commands to replace the CR by proper Linux newline
EMACS_EXPR1='(replace-string  "\r" "" nil nil nil)'
EMACS_CMD=emacs -q --no-site-file -batch
EMACS_OPTS=--eval=$(EMACS_EXPR1) -f save-buffer
```

5.4 Physical Layout of the File

The the physical layout of the file is simply the juxtaposition of the the above defined elements:

41 ⟨*Makefile* 41⟩≡
  ⟨*makefile variables* 34⟩
  ⟨*emacs-commands* 40c⟩
  ⟨*abstract-targets* 36⟩
  ⟨*detailed-targets* 37⟩
  ⟨*make-Makefile.0* 33c⟩

5.5 Makefile History

2006 03 11: GF update to make install sal.

2006 03 12: GF update to add paths for sal installation. added command to fix the newlines in the lisp output.

2006 03 14: GF created the noweb version of this file.

2006 03 15: GF update to add automated testing.

2006 03 16: GF draft version of descriptive text completed. Some minor updates to streamline and homogenize the code.

2006 05 23: GF added headings "-12345" option to HTML_TOC command to enable generation of all headers in the HTML document table of contents.

2006 03 31: GF added "test-no-quit" targets to allow the user to visit the lisp envt. after running test harnesses.

2006 05 30: GF changed code mode to Makefile-mode.

2006 05 31: GF updated to include making of the distribution archive.

2006 06 07: GF update after final review.

Chapter 6

SAL: The Top level

This is a description of the top level API to SAL. It is provided as a lisp source file **sal.lisp** generated from the literate file **sal.nw**.

We will examine the functionality provided in this file following the order of use, not the layout of the file itself.

6.1 The SAL Package

The SAL package is the starting point for all SAL's functionality. Loading this package loads all the other packages and initializes the system. The package itself provides the functionality as per the following definition:

42a ⟨*sal-package* 42a⟩≡ (59)

```
(defpackage "SAL"
  ;; provide a package to encapsulate the database functionality
  (:use "COMMON-LISP")
  (:export "REPORT"
           "PROCESS"
           ))

(in-package "SAL")
```

As can be seen, there are only two exported symbols in the SAL package. These are all functions and are described below.

6.2 API Functionality

The SAL API provides two services. A general processing service and a specialized reporting service.

42b ⟨*sal-api* 42b⟩≡ (59)

⟨*report* 47a⟩
⟨*process* 43a⟩

6.2.1 process (<args>)

This function is the main api call provided by SAL. It performs all loading, initialization, input and output for automated processing. A very detailed explanation of the use of this function is provided in the How-To in chapter 4.3.2 on page 10.

Arguments: *The arguments are fully described in the How-To.*
Return:

- The resulting internal database is returned if processing is successful.
- In case of failure, an error is logged and *nil* is returned.

The function `process` looks like this:

43a ⟨*process* 43a⟩≡ (42b)
  ⟨*sal:process-arg-def* 12⟩
  ⟨*sal:process-body* 43b⟩

We'll now look at what happens during the execution of the body of `process`.
Firstly, a call to `log-it` is used to log a *start of processing* message.

43b ⟨*sal:process-body* 43b⟩≡ (43a) 44 ▷

```
(wut:log-it
 (format nil "Sal-Process: processing ticker: ~S" ticker-string)
 logging)
```

Then, wrapped in an *unwind-protect* is a *let* which does it all:

1. The local variable *db* is initialized by a call to `init-sugar`, which also initializes the *sugar functions* module,
2. The *rule functions* module is initialized with a call to its initialization function `rf:init`.
3. A datum is defined to ensure that there is a default value for the "INDUSTRY" attribute,
4. `defdata` is called with the *data* argument, loading all the data, but skipping any invalid elements. If no valid data is loaded, then an error is raised and the *unwind-protect* will step in to execute a log-it before exiting,
5. The file indicated in the *mdl-path* argument is loaded,
6. The rules are loaded via a call to `load-rules`,
7. If a *report-start* and *report-stop* date have been provided as arguments, then the function `set-up-and-report` is called. This does just what its names says: set-up for report generation and generate,
8. Finally, the *end of processing* is logged.
9. At the end of the function, we see that in case of error in the *let, unwind-protect* will execute a logging call before returning *nil*.

So we see that a whole lot of functionality is hidden away in the embedded calls. How does it really work? Well just as we said, most of the work is handled in the initialization of the component modules. This includes, creating and populating the internal database with data and rules. Once that is done, all that remains is to perform the queries required to get the reports.

It's a simple as that!

44 ⟨*sal:process-body* 43b⟩+≡ (43a) ◁43b

```
(unwind-protect
    (let ((db (sgr:init-sugar ticker-string
                              current-year
                              :loop-detect loop-detect)))
      (rf:init db)
      ;; create a default industry, just in case!
      (rf:defdata
       (list "INDUSTRY" (cfg-pkg-name-eval "*sm-default-industry*")))
      (when (zerop (rf:defdata data))
        (error "Sal-Process: no data to load!~%Data is: ~S" data))
      (load mdl-path :verbose verbose :print verbose)
      (load-rules :verbose verbose)
      (when (and report-start report-stop)
        (set-up-and-report db report-start report-stop
                           std-out file-out out-file-name))
```

```
    (wut:log-it
     (format nil
             "Sal-Process: processing completed! ticker: ~S"
             ticker-string)
     nil)
    db)
(wut:log-it
 (format nil "Sal-Process: Processing  Failed! ticker: ~S" ticker-string))
()))
```

6.2.2 `report` (<args>)

Arguments:

:start first year to report,

:stop last year to report,

:att-lis a list of strings containing the attributes to report,

:o-stream-lis is a list of open streams for writing, defaults to `(t)`, meaning std-out. Output will be written to all the streams in the list.

Return:

`t` .

This function reports to a stream previously opened for writing. The report format is line-based and comma-separated. The first line contains headers. Each of the lines following the header contains the values that correspond to the headers.

The following example illustrates a report. Suppose that we have loaded attributes for the stock "IBM", with ID value "123". Suppose that `report` is called with the arguments:

:start = 2005

:stop = 2006,

:att-lis = `("Profit "Losses")`

The following output would be sent to std-out (this is the default value for the output stream argument):

```
"Ticker","ID","Attrbiute",yr_2005,yr_2006
"IBM",123,"Profit",1000,20000
"IBM",123,"Losses",500,3000
```

The function `report` is relatively simple to understand. It is based on mapping output production over the output streams. It relies on a helper functions `report-header` to create the header line and `report-line` in collaboration with the well known *sugar function queries* to build the output data lines. The rest is just straight Lisp.

Here's a run though:

1. First, if there are no output streams, do nothing.
2. If there are outputs streams, then create the header line and map it via a **format** statement to all the output streams,
3. then, for each attribute, do the same mapping for the report-lines created by feeding a *sugar query* for the ticker to `report-line`. The embedded call to `report-line` performs the *sugar query* for the attribute values on $[start, stop]$,

4. last of all, close all the out-streams that are not std-out,
5. and return **t**.

47a ⟨*report* 47a⟩≡ (42b)

```
(defun report (&key start stop att-lis (o-stream-lis (list t)))
  ;;(when sal-trace
  ;;(format t "sal:report: att-lis: ~S~%" att-lis)
  ;;(break))
  (if o-stream-lis
      (progn
        (let ((header (report-header start stop)))
          (mapc #'(lambda(o-stream)
                    (format o-stream header))
                o-stream-lis))
        (mapc
         #'(lambda(att)
             (let ((line
                    (report-line start
                                 stop
                                 (sgr:apply-attribute-sugar "TICKER")
                                 att)))
               (mapc #'(lambda(o-stream)
                         (format o-stream line))
                     o-stream-lis)))
             att-lis)
        (mapc #'(lambda(o-stream)
                  (when (not (eq o-stream t))
                    (close o-stream)))
              o-stream-lis))
    t))
```

6.3 API Helper Functions

These functions provide the first level of support to SAL's API. They are physically organized as per the following chunk.

47b ⟨*api-helpers* 47b⟩≡ (59)

⟨*report-reduce-helper* 51a⟩
⟨*report-header* 49⟩
⟨*report-line* 50⟩
⟨*set-up-and-report* 48⟩

They will be described in the hierarchical order in which they are called.

6.3.1 set-up-and-report (<args>)

Arguments:

db : an internal database,

report-start : report start date as absolute year,

report-stop : report stop date as absolute year,

std-out : boolean if TRUE report to std out

file-out :boolean if TRUE report to file out

out-file-name : file name as string for file output, note: ".csv" will be appended to the name.

Return:

t the result of the call to `report`.

This function does the preparation needed to call `report`. It simply sets up the call with the information provided by `process`. It assumes that both report-start and report-stop are valid numbers, absolute year values. No checking is performed.

There are a some things worth noting in the body of this function.

1. The first is the call to the `get-model-attributes`. This function, provided by the **internal-database** package returns the current list of model-attributes. The keyword argument **:report?** allows for the selection of *reporting* attributes or *all* of the model attributes.

2. The second is the construction of the file output stream. The name is built either from the supplied argument **out-file-name** if provided, or if not provided by the creation of a name using the *ticker string.* For example, if the ticker were *"IBM"*, then the out-file-name would be **IBM.csv.** Remember, the out-file-name argument is always used if provided.

3. Finally, one should note that all file writing is done according to the configuration parameter `*io-output-path-string*`.

48 ⟨*set-up-and-report* 48⟩≡ (47b)

```
(defun set-up-and-report (db report-start report-stop std-out file-out
                             out-file-name)
  (report :start report-start
          :stop report-stop
          :att-lis
          (ids:get-model-attributes :ht db :report? t)
          :o-stream-lis
```

```
            (append (when std-out
                      (list t))
                    (when file-out
                      (list (wut:out-stream
                             :file-name-string
                             (concatenate
                              'string
                              (or out-file-name
                                  (sgr:apply-attribute-sugar "TICKER"))
                              ".csv")
                             :path-string
                             (cfg-pkg-name-eval "*io-output-path-string*")))))))
```

6.3.2 `report-header` (start stop)

This function builds and returns the report header for use by `report` function.

Arguments:

1. start year for report as number,
2. stop year for report as number.

Return multiple values:

- A string containing the official report headers:
  Ticker, ID, Field, yr_1, yr_2, …

49 ⟨*report-header* 49⟩≡ (47b)

```
  (defun report-header(start stop)
    (concatenate 'string
                 "\"Ticker\",\"ID\",\"Field\","
                 (if (= start stop)
                     (format nil "\"yr_~A\"" start)
                   (reduce #'(lambda(l r)
                               (multiple-value-bind
                                (fix-l fix-r)
                                (report-reduce-helper l r)
                                (format nil "~A,\"~A\"" fix-l fix-r)))
                           (wut:numlist start stop)))
                 "~%"))
```

6.3.3 `report-line` (start stop ticker att)

This function builds and returns the report lines for use by `report` function.

Arguments:

1. start year for report as number,
2. stop year for report as number.
3. ticker as string
4. attribute as string

Return multiple values:

- A string containing the line:
  ticker-value, ID-value, att-name, att-val-1, att-val-2, ...

50 ⟨*report-line* 50⟩≡ (47b)

```
(defun report-line(start stop ticker att)
  (concatenate 'string
               ;; ticker,id,att-name
               (format nil "~S,~S,~S,"
                       ticker
                       (rational (sgr:apply-attribute-sugar "ID"))
                       att)
               ;; val-1,val-2...
               (if (= start stop)
                   ;; only one val, don't use reduce!
                   (format nil "~F"
                           (car
                            (sgr:apply-attribute-sugar att
                                                       start
                                                       stop)))
                 ;; at least 2 vals, use reduce to make list
                 (reduce #'(lambda(l r)
                             (format nil "~F,~F" l r))
                         (sgr:apply-attribute-sugar att
                                                    start
                                                    stop)))
               "~%"))
```

6.3.4 `report-reduce-helper` (l r)

This function is used as support for a call to reduce. It assumes that the arguments will be such that at least the second is a number. It will return appropriate values for header columns, i.e. "some string" and "yr_NNNN".

Arguments:

1. any lisp object, but only a string or a number is appropriate,
2. any lisp object, but only a number is appropriate.

Return multiple values:

1. if first argument was a number, "yr_NNNN", if not the argument is returned,
2. the second argument prepended with "yr_".

51a ⟨*report-reduce-helper* 51a⟩≡ (47b)

```
(defun report-reduce-helper (l r)
  "takes the arguments give by reduce and fixes them with yr_.
Returns 2 values"
  (let ((fixed-l (if (numberp l) (format nil "\"yr_~A\"" l) l))
        (fixed-r (format nil "yr_~A" r)))
    (values fixed-l fixed-r)))
```

6.4 Local Loading Utility Functions

These functions and commands provide loading support to SAL. They are physically organized as per the following chunk.

51b ⟨*sal-loading-utils* 51b⟩≡ (59)

⟨*load-sal* 52⟩
⟨*call-load-sal* 53a⟩
⟨*src-name-2-bin-name* 56b⟩
⟨*path-bin-it* 56a⟩
⟨*key-2-pathname-lis* 55⟩
⟨*load-rules-helper* 54⟩
⟨*load-rules* 53b⟩

They will be described in the hierarchical order in which they are called.

6.4.1 `load-sal` (cfg-pathname &key (verbose t))

This function loads all the compiled SAL files as per the config path.

Arguments:

1. Full pathname object, NOT the STRING path, to the config file,
2. **verbose:** if TRUE, load verbosely, otherwise load silently.

Return:

- The list of file-names of the files loaded.

The body of this function performs the following operations:

1. load the config file to gain access to the various configuration information needed to proceed,
2. in a *let**, first obtain the list of source file names. It is of note that the function `cfg-pkg-name-eval` is used to translate a string into the value of the variable named by the string. This technique is used to avoid reference to unknown packages when the sal source or binary file is first loaded.
3. next, the compiled file-names are built from the source file-names, excluding "sal" and "sal-build" which are not needed,
4. once the binary file-names are available, the function *load* is simply mapped over them passing the variable `verbose` twice, to get either really verbose or really silent output.

52 ⟨*load-sal* 52⟩≡ (51b)

```
(defun load-sal(cfg-pathname
               &key
               (verbose t))
  (load-config cfg-pathname)
  (let* ((src-name-lis (cfg-pkg-name-eval "*bi-src-filename-string-list*"))
         (compiled-file-names
          (mapcar #'(lambda(src-name)
                      ;; find '.' replace after with bin extension
                      ;; src-name
                      (concatenate
                       'string
                       (subseq src-name 0 (search ".lisp" src-name))
                       (cfg-pkg-name-eval "*bi-bin-extension-string*")))
                  (remove "sal.lisp"
                          (remove "sal-build.lisp"
                                  src-name-lis
                                  :test #'equal)
                          :test #'equal))))
```

```
      (mapc #'(lambda(f-name)
                (load (pathname
                       (concatenate
                        'string
                        (cfg-pkg-name-eval "*bi-sys-path-string*")
                        f-name))
                      :verbose verbose
                      :print verbose))
            compiled-file-names)))
```

After defining the function `load-sal` we call it directly from the top level of the file. As the file "sal" is read into lisp, when it reaches this point, it performs this call, and loads all of the SAL files.

53a ⟨*call-load-sal* 53a⟩≡ (51b)

```
;;; this loads it all!
(load-sal (pathname "⟨configfile-path 3a⟩")  :verbose t)
```

6.4.2 `load-rules` (&key (verbose t))

This function loads the default rules, then applies `sgr:industry` to get the industry specific rules and loads them. All other data comes from config.

Arguments:

verbose: if TRUE, load verbosely, otherwise load silently.

Return:

- List or pathnames of specific rule files loaded if success,
- *nil* if failure to find industry specific rule files, defaults should always load!

53b ⟨*load-rules* 53b⟩≡ (51b)

```
(defun load-rules (&key (verbose t))
  ;; load the default rules
  (load-rules-helper t :verbose verbose)
  ;; get and load the industry specific rules, if there are any...
  (let ((ind (sgr:apply-attribute-sugar "INDUSTRY")))
    (if ind
        (load-rules-helper ind :verbose verbose)
      (wut:log-it (format nil "No rules for industry: ~S" ind)))))
```

6.4.3 `load-rules-helper` (key &key verbose)

This function loads the rule-file associated with key. It uses the config to find the rest of the data needed to perform the load. Loading and loaded filenames are sent to log.

Arguments:

1. a string that will be used as key to lookup the rule-file names in the config data,
2. **verbose:** if TRUE, load verbosely, otherwise load silently.

Return:

- A list of pathname objects if success,
- NIL if failure.

The body of this function performs the following operations:

1. Set up a let to get the list of pathname objects corresponding to rule-files that are associated with the argument *key,*
2. Map over the list, loading each file, logging as we go,
3. If there are no rules to load, this is logged, too.

54 ⟨*load-rules-helper* 54⟩≡ (51b)

```
(defun load-rules-helper (key &key verbose)
  (let ((pathname-lis
         (key-2-pathname-lis key
                             (cfg-pkg-name-eval
                              "*sm-industry-rulefile-string-alist*")
                             (cfg-pkg-name-eval "*sm-model-path-string*"))))
    ;;(when sal-trace
    ;;(format t "pathname-lis: ~S~%" pathname-lis)
    ;;(break))
    (if pathname-lis (mapc #'(lambda(path)
                               (wut:log-it
                                (format nil
                                        "load-rules-helper: loading: ~S"
                                        path))
                               (load path :verbose verbose :print verbose))
                           pathname-lis)
      (wut:log-it (format nil "No rule-file for key: ~S" key)))))
```

6.4.4 `key-2-pathname-lis` (key a-lis path-string)

This function returns a list of pathnames for binary files as associated to key in a-lis. The a-lis values are source file names. These names are manipulated by the function so as to return binary filenames. All arguments must have types, i.e. STRING, that correspond correctly to those of a-lis.

Arguments:

1. a key as a string,
2. an a-lis such that key is string or list of strings, values are string or list of strings that correspond to file-names with .lisp extensions.
3. A string representation of a path to the file.

Return:

- if an assoc is found, A list of pathname objects resulting from the concatenation of the path-string and the file-name with bin extension.
- if not, nil.

55 ⟨*key-2-pathname-lis* 55⟩≡ (51b)

```
(defun key-2-pathname-lis (key a-lis path-string)
  (let ((res (cdr (wut:s-assoc key a-lis))))
    ;;(when sal-trace
    ;;(format t "key lookup: ~S~%" res)
    ;;(break))
    (if (null res) ()
      (path-bin-it :file-or-lis res
                   :path-string path-string)))))
```

6.4.5 `path-bin-it` (&key file-or-lis path-string)

This function returns a list of pathnames to binary files as built from the arguments and from config data.

Arguments:

file-or-lis a string or list of strings taken to be **file-name.something**, NOTE only the dot **"."** counts for the concatenation of the bin extension,

path-string a string path.

Return:

- list of pathnames with binary extensions.

The body of this function performs the following operations:

1. If called with list of files, then map a recursive call over each of them,
2. Otherwise, build and return a list of the binary filenames

56a ⟨*path-bin-it* 56a⟩≡ (51b)

```
(defun path-bin-it (&key file-or-lis path-string)
  (if (listp file-or-lis)
      (mapcan #'(lambda(file-name)
                  (path-bin-it :file-or-lis file-name
                               :path-string path-string))
              file-or-lis)
    (list
     (wut:path-get
      (src-name-2-bin-name file-or-lis
                           ".lisp")
                       path-string))))
```

6.4.6 `src-name-2-bin-name` (src-name bin-ext)

This function returns the **"name.lisp"** changed to **"name.bin-ext"**.

Arguments:

1. a string file name of form **xxxx.yyy**,
2. a string NOT including the **"."** to be concatenated at the end.

Return:

- The string "name.bin-ext".

56b ⟨*src-name-2-bin-name* 56b⟩≡ (51b)

```
(defun src-name-2-bin-name (src-name bin-ext)
  (concatenate 'string
               (subseq src-name 0 (position #\. src-name))
               bin-ext))
```

6.4.7 Misc.

The following dummy package definitions are needed to ensure that SAL can pass an initial parse before the real definitions of its required packages are available.

57 ⟨*loading-astuces* 57⟩≡ (59)

```
(defpackage "SAL-CONFIG"
  (:nicknames "SAL-CFG"))

(defpackage "WIG-UTIL"
  (:nicknames "WUT"))

(defpackage "SUGAR"
  (:nicknames "SGR"))
```

6.5 Test Harness

58a ⟨*sal-test-harness* 58a⟩≡ (59)

```
(defparameter *c-c-alis* ())
(setf *c-c-alis*
  '(("Go for an analysis on [2005 2010],
     verbose,
     print to std-out"
    .
    "(format t \"~S~%\" (sal:process :ticker-string \"ibm\"
                                     :current-year 2006
                                     :verbose t
                                     :data *ibm-data*
                                     :report-start 2005
                                     :report-stop 2010
                                     :std-out t
                                     :file-out t
                                     :out-file-name \"this-is-output\"
                                     :logging t
                                     :loop-detect t))")
    ("Go for an analysis on [2005 2010],
     NOT verbose, using default current-year
     print to std-out"
    .
    "(format t \"~S~%\" (sal:process :ticker-string \"ibm\"
                                     :verbose nil
                                     :data *ibm-data*
                                     :report-start 2005
                                     :report-stop 2010
                                     :std-out t
                                     :file-out nil
                                     :logging t
                                     :loop-detect t))")
    ))
```

The following items are included to make loading and testing of SAL easier during debugging.

58b ⟨*sal-debugging-helpers* 58b⟩≡ (59)

```
;;; helper comments to ease loading!
;(load "/home/bob/Desktop/Programming/lisp/StockEvaluator/Work/V6.9.1/sal.lisp")
;(load "/home/bob/Desktop/Programming/lisp/StockEvaluator/Work/Examples/data.lisp")
;(load "/home/bob/Desktop/Programming/lisp/StockEvaluator/Work/bin/sal.fas")
;(load "/home/bob/Desktop/Programming/lisp/StockEvaluator/Work/bin/sal.x86f")

;;(defparameter sal-trace ())
```

6.6 Physical Layout of the File

The package is ordered as per the following:

59 ⟨*sal.lisp* 59⟩≡
```
;;; sal.lisp
⟨lisp-header 143b⟩
⟨sal-debugging-helpers 58b⟩
⟨loading-astuces 57⟩
⟨sal-package 42a⟩
⟨load-config 155⟩
⟨cfg-pkg-name-eval 130a⟩
⟨sal-loading-utils 51b⟩
⟨api-helpers 47b⟩
⟨sal-api 42b⟩
⟨eoc 143c⟩
⟨sal-test-harness 58a⟩
⟨eof 144⟩
```

6.7 Sal Package History

2006 01 23: GF creation of the file.

2006 01 24: GF creation of load-config, make-install, sal.

2006 01 29: GF first working version!

2006 02 01: GF made the test harness *C-C-ALIS* private to the SAL package.

2006 02 03: GF minor updates after code-review. add check for empty data loading. update to add logging control argument to sal:process. preparation for tests of loading compiled rules.

2006 02 06: GF update to add :print verbose argument to all load commands.

2006 02 07: GF update process arguments to support call to init-sugar for loop detection. Commented out all tracing. Added dummy package defs for WIG-UTIL and SGR to satisfy the CMUCL compiler...

2006 02 08: GF modify key-2-pathname-lis to use wut:s-assoc instead of ordinary assoc; this implements multiple industry ¡-¿ rule-file associations.

2006 02 09: GF update process, report to remove ticker argument to report, creation of set-up-and-report to clean up process and report which is made more efficient. Suppression of all reference to dbs, since this is an obsolete concept.

2006 02 09: GF update process & set-up-and-report to add out-file-name argument. corrected load-rules to log message BEFORE loading begins, not AFTER it is complete.

2006 03 12: GF update to make configfile pathname argument to load-sal, and to eliminate the need for preloading of sal-config file.

2006 03 30: GF correct report-reduce-helper and report-header so that yr_2005 will appear as "yr_2005". thus all the strings are output enclosed in double quotes.

2006 05 01: GF remove unused *sal-startup-message* *sal-help-message*

2006 05 03: GF change the last call to log-it in sal:process to not specify second argument, thus allowing logging to continue as it was.

2006 05 07: GF Remove "SET-UP-AND-REPORT" from the public interface. This should have been done long ago; the function is not needed by a user.

Chapter 7

The Sugar Functionality

The *sugar functionality* is the architectural heart of SAL. Nearly all user level functionality is built upon it. For memory, we call *sugar function* the syntactic sugar that allows the user to write:

```
(profit -1)
```

when he is really saying:

```
select profit
from data-table
where profit.year = (1- current-year).
```

In other words, *sugar functions* encapsulate database queries.

But where do they come from? Where are they stored? How do they work? Where do they go when no longer needed?

Oh so many questions, and so few answers ...

Before we try to answer, let's look at the structure of the Sugar package.

7.1 The SUGAR Package

The SUGAR package provides functionality as per the following definition:

61 ⟨*sugar-package-def* 61⟩≡ (98b)

```
;; provide a package to encapsulate the sugar functions.
(defpackage "SUGAR"
  ;; this pkg will access std lisp functions, only.
  (:use "COMMON-LISP")
  (:nicknames "SGR")
  (:export "SUGAR-FUNCTION-NAME"
           "SUGAR-FUNCTION-SYMBOL"
           "ATTRIBUTE-SUGAR-FUNCTION"
```

```
            "APPLY-ATTRIBUTE-SUGAR"
            "ATT-NAME-2-SUGAR-FUNC"
            "INDUSTRY"
            "INIT-SUGAR"
            "MODEL-ATTRIBUTES"
            ))

(in-package "SGR")
```

As can be seen above, the Sugar package offers quite a few public symbols. We'll look at them all in due time.

7.2 Where do Sugar Functions come from?

Since a *sugar function* is needed for each attribute in the system, the system creates a *sugar function* each time it meets a new attribute. This happens each time an attribute is introduced to SAL by means of `defdatum`, `defmodel` or `defreport`. Each of these functions embeds a call to `att-name-2-sugar-func`. This is the entry point for the creation of sugar functions.

7.2.1 `att-name-2-sugar-func` (att-name db &optional current-year)

This function is called each time SAL is looking for the sugar function that is associated with the attribute named in the first argument. The function will always return the corresponding *#'function*. It uses delegation to perform the actual work. If the attribute is already known, then a call is made to `attribute-sugar-function` which returns the *sugar function* requested. If the attribute is unknown, then a *sugar function* is created by a call to `create-sugar-function`.

One may take note that this function accepts the special attribute names *t* and *nil*. Indeed, *t* is special in that it first *translates* into the string "MODEL-ATTRIBUTES" with the obvious signification. The special attribute *nil* translates into the string "GENERAL-RULES". These are further discussed as they appear in the code below.

Arguments:

1. the attribute name as a string, or a special attribute name: *t* or *nil*,
2. an internal database containing all known data,
3. optional current year needed in case of *sugar function* creation.

Return:

- The *#'sugar-function*. If this function ever returns *nil* then there is a serious problem with the database or data itself.

63 ⟨*att-name-2-sugar-func* 63⟩≡ (98c)

```
(defun att-name-2-sugar-func (att-name db &optional current-year)
  ;; if att-name is a member of the keys, or of the model attributes,
  ;; then the corresponding sugar function should exist.
  (if (or (member att-name
                  (ids:get-keys db)
                  :test #'equal)
          (member att-name
                  (ids:get-model-attributes :ht db)
```

```
                :test #'equal))
      (attribute-sugar-function att-name)
    (create-sugar-function att-name db current-year)))
```

If an attribute is already known to SAL, then finding its *sugar function* based on its string name is performed by `attribute-sugar-function`:

7.2.2 `attribute-sugar-function` (att-str)

This function will return the *#'sugar-function* corresponding to the argument. If there is no such function then *nil* will be returned. The value of the argument is not used directly, it is first used to find the `sugar-function-symbol` which is in turn used to find the *sugar function.*

Arguments:

1. the attribute name as a string, or a special attribute name: *t* or *nil*,

Return:

- The *#'sugar-function* or *nil* if not found.

64a ⟨*attribute-sugar-function* 64a⟩≡ (98c)

```
(defun attribute-sugar-function (att-str)
  (let ((sym (sugar-function-symbol att-str)))
    (if sym (symbol-function sym)
      ())))
```

The next part of the indirection on the attribute's name is described in the following section.

7.2.3 `sugar-function-symbol` (obj)

This function takes the lisp object provided as argument and looks up the symbol in the "SUGAR" package that corresponds to the `sugar-function-name` of the object. The argument can be a string, *t*, or *nil*. There is no creation of sugar if the function is missing. In the latter case *nil* is returned.

Arguments:

1. an object that could be a string, *t* or *nil*,

Return:

- The symbol corresponding to the function, or *nil* if not found.

64b ⟨*sugar-function-symbol* 64b⟩≡ (98c)

```
(defun sugar-function-symbol (obj)
  (find-symbol (sugar-function-name obj) "SUGAR"))
```

7.2.4 `sugar-function-name (obj)`

This function takes an object that can be *t*, *nil* or a string such as “Free Cash Flow” and returns a new string such as “FREE-CASH-FLOW” removing all leading space and multiple space before replacing single space by “-” and converting all characters to uppercase.

In the case of *t* the string returned is “MODEL-ATTRIBUTES” and in the case of *nil* is “GENERAL-RULES” is returned.

Strings with embedded hyphens “-” will be *rejected* and an error will be signaled.

Arguments:

1. an object that could be a string, *t* or *nil*.

Return:

- The string corresponding to the sugar function name of the object.

65 ⟨*sugar-function-name* 65⟩≡ (98c)

```
(defun sugar-function-name (obj)
  (cond
   ((null obj) "GENERAL-RULES")
   ((eq t obj) "MODEL-ATTRIBUTES")
   ((or (not (stringp obj))
        (find #\- obj))
    (error "sugar-function-name: Cannot make sugar-function-name from ~S~%"
           obj))
   (t  (string-upcase
        (substitute #\-
                    #\Space
                    (remove-double-spaces obj)
                    :test #'char=)))))
```

Now that we've seen how to find an existing *sugar function,* let's look at how to create a new one!

7.2.5 `create-sugar-function` (att-name db current-year)

Well the truth is that there isn't much to it. The idea is that we first figure out a symbolic name for the *sugar function*, then if the name is not already in use by Common Lisp, we create a lambda expression, i.e. the function body, and set the symbol's function value to that lambda expression. That's all there is to it!

The lambda expression is a closure containing a function of unspecified arguments which simply delegates the processing to `exec-sugar` while encapsulating the values of the attribute name, a pointer to the internal database, and the current year.

Once all the assignment is complete, the symbol is exported from the SUGAR package and returned for good luck!

NOTE: In case of name collision between the symbol to which the *sugar function* should be assigned and a symbol in the "Common-Lisp" package, then an error is raised and processing is aborted.

Arguments:

1. a string from which the sugar function's symbol name will be generated,
2. an internal database,
3. the current year.

Return:

- the symbol to which the *sugar function* has been assigned, exported from the "SUGAR" package.

66 ⟨*create-sugar-function* 66⟩≡ (98e)

```
(defun create-sugar-function (att-name db current-year)
  (let ((sugar-func-name (sugar-function-name att-name))
        (cur-yr (if current-year
                    current-year
                  (funcall (attribute-sugar-function "CURRENT YEAR")))))
    (if (find-symbol sugar-func-name "COMMON-LISP")
        (error "The sugar function name ~S is in use by COMMON-LISP."
               sugar-func-name)
      (let ((sym (intern sugar-func-name (find-package "SUGAR")))
            (func #'(lambda(&rest args)
                      (exec-sugar att-name
                                  args
                                  db
                                  cur-yr))))
        (setf (symbol-function sym) func)
        (export sym "SUGAR")
        sym))))
```

We've seen all the embedded calls in `create-sugar-function` except for the most important `exec-sugar`. This one, and its friends will be explained now (aren't just sitting on the edge of your seat, holding your breath . . .)

7.3 How do Sugar Functions Work?

Well they say that expectation is the true pleasure in life. If that is the case then the explanation of *sugar functions* should create immense pleasure for my dear readers. It is long but not complex.

The idea is that sugar calls are parsed according to the length of the argument list which they receive. For each argument list length, specific conditions apply and actions must be taken. These are performed by specific parsing functions, conveniently named `parse-0`, `parse-1`, etc. where the number following the hyphen indicates the length of the argument list that is parsed.

The first step in the parsing is the execution of the function `exec-sugar`.

7.3.1 `exec-sugar` (att-name args ht cur-yr &optional (f-lis *parse-funcs*))

This is the first step in executing a sugar call. In fact, this is a dispatcher function that takes the length of the sugar call's argument list as an index to a list of parsing functions and dispatches appropriately.

Error checking is performed on the length of the argument list, not on the content. If there are too many arguments, then an error is raised and processing is aborted.

The arguments to `exec-sugar` are all passed by encapsulation from `create-sugar-function`, except the optional list of parsing functions for which the default value is always used.

Arguments:

1. the attribute name from `create-sugar-function`,
2. the arguments passed to the sugar call from `create-sugar-function`,
3. the internal database, here called a hash-table since that is what it is in this implementation, from `create-sugar-function`,
4. the current year, again from `create-sugar-function`,
5. the list of parsing functions indexed on the length of the argument list that the function parses.

Return:

- The result of the sugar call is returned, whatever that may be.

68a ⟨*exec-sugar* 68a⟩≡ (99c)
```
(defun exec-sugar (att-name args ht cur-yr &optional (f-lis *parse-funcs*))
  "This is the dispatcher & work initiator for all sugar functions.
It dispatches according to the length of the argument list supplied to
the sugar function. Error checking is present.
"
  (let ((ind (length args))
        (err-msg
         (concatenate
          'string
          "Too many arguments to ~S sugar function call!~%"
          "Expected at most ~S, received ~S: args: ~S")))
    (if (>= ind (length f-lis))
        (error  err-msg att-name (1- (length f-lis)) ind args)
      (funcall (nth ind f-lis) att-name args ht cur-yr))))
```

The parsing functions are named *parse-n* where n indicates the length of the argument list that the function will parse.

68b ⟨*parse-funcs* 68b⟩≡ (99c)
```
(defparameter *parse-funcs*
  (list
   #'parse-0
   #'parse-1
   #'parse-2
   #'parse-3
   #'parse-4))
```

The sugar functions are the user API to the internal data structures. They are created from attribute-names as returned from the utility function `sugar-function-name`. There can be no name collisions since the SUGAR package detects and rejects creation of sugar-functions that would collide with existing functions.

The sugar functions are specified as:

```
; (sugar (&rest args))
;   returns 2 values depending on the arguments
;   all sugar functions query unless one of the keywords
;   :set
;   :project
;   :model
;   :rule
;   is present,
;   in which case assignemnt is performed.
```

We will now describe the parsing functions one by one.

7.3.2 Arguments to the Parsing Functions

Since all the parsing functions are applied to the same argument list, they all take the same arguments. The processing of each of these differs and is described with the particular functions.

Parse Function Arguments:

1. **att-name**, the attribute name as a string in its original form and case sensitive,
2. **date value**, t for a fact, absolute or relative year otherwise,
3. **ht**, the internal database,
4. **cur-yr**, the current year's value,
5. optional **accept-projected?** default to t implying that projected values will be accepted as valid for the return, if *nil* is used here then projected values are filtered out and replaced by *nil*.

7.3.3 `parse-0`

This is the case of the arg list of length 1, e.g. **(sugar).**

This case is equivalent to:

```
(or (sugar t)
    (sugar current-year))
```

The processing returns the result of the first successful execution of a rule or *nil*; *nil* if all fail. Rules may project into the db. In this case, we first attempt to get the fact value, then the yearly value for current year. It is possible to filter on "accept-projected?", as needed for potential internal calls to this function. The actual execution of the query is delegated to `parse-1`.

Arguments as per all parsing functions.

Return:

1. the value of sugar's factual value, or value for current year if not a known fact; or *nil* if none found,

2. *t* if the value is the result of a projection, *nil* if not projected or if no value was found,

70 ⟨*parse-0* 70⟩≡ (99a)

```
(defun parse-0 (att-name unused ht cur-yr &optional (accept-projected? t))
  (declare (ignore unused))
  ;;(when sgr-trace
  ;;(format t "Parse-0:~%att-name: ~S~%Current year: ~S~%"
  ;;    att-name cur-yr)
  ;;(break))
  (parse-1 att-name (list cur-yr) ht cur-yr accept-projected?))
```

7.3.4 `parse-1`

This is the case of the arg list of length 0, e.g. **(sugar t) (sugar -2) (sugar 1998) (sugar ()).**

The single argument may be any of:

- *t* implying we are seeking a fact value,
- a relative or absolute year,
- *nil* implying that we are looking for rules.

The processing returns the result of the first successful execution of a rule or *nil*; *nil* if all fail. Rules may project into the db. In this case, we first attempt to get the fact value, then the yearly value for current year. It is possible to filter on "accept-projected?", as needed for potential internal calls to this function.

Arguments as per all parsing functions.

Return:

- if the argument is *t*:
  1. sugar's fact value, or nil if not known and not projectable,
  2. *t* if the value is the result of a projection, *nil* otherwise.
- if the argument is an absolute or relative year value:
  1. sugar's fact value if known or projectable; or sugar's value for the year if known or projectable; *nil* if neither are known or projectable.
  2. *t* if the value is the result of a projection, *nil* otherwise.
- if the argument is *nil* or if the att-name is *nil*:
  1. all rules that are associated with **sugar**, inclusive of general rules, in the order of increasing precedence,
  2. only specific rules that are associated with **sugar**, in the order of increasing precedence.

One subtlety of this and of many of the parsing functions the use of the lisp function **destructuring-bind.** This function uses the Common-Lisp parser to match key-words from an argument list, to local variables in the code. It is very useful indeed.

The function `parse-1` proceeds as follows:

1. First, bind the single argument to the local variable *date,*
2. if *date* is null, then delegate to the database function `get-rules` to get the rules.

3. otherwise, bind the values obtained by a delegated call to `find-or-project` for the fact value,
4. if this returns a non null value, then it is returned,
5. if not, and if date is a year, then again delegate to get the timely value,
6. otherwise, give up and return the failure values.

72 ⟨*parse-1* 72⟩≡ (99a)

```
(defun parse-1 (att-name arg-lis ht cur-yr &optional (accept-projected? t))
  (destructuring-bind
   (date) arg-lis
   ;;(when sgr-trace
   ;;(format t "Parse-1:~%att-name: ~S~%Current year: ~S~%Date: ~S~%"
   ;;      att-name cur-yr date)
   ;;(break))
   (if (null date)  ; return the rules
       (ids:get-rules att-name :ht ht)
     (multiple-value-bind
      (val projected?)
      (find-or-project att-name
                       t
                       ht
                       :accept-projected? accept-projected?)
                       ;; CORRECTION 2006 04 30 nil)
      (cond
       (val (values val projected?))
       ((not (eq date t))
        (find-or-project att-name
                         (wut:abs-year date cur-yr)
                         ht
                         :accept-projected? accept-projected?))
       (t (values ()()))))))))
```

7.3.5 `parse-2`

If argument list is of length 2, there are three cases: two queries, one assignment.

Case-1: Query for a period, e.g. **(sugar -2 2010)**.

In this case the arguments are *start-year, stop-year* and these may be absolute or relative.

The return values are:

1. a list of sugar's values for each year on *[start-year, stop-year]*, or *nil* if not known and not projectable,
2. a list with a value for each year that is *t* if the value is the result of a projection, *nil* otherwise.

Case-2: Query for *t* or current-year while filtering projected values, e.g. **(sugar :projected? nil)** In this case the arguments are the keyword ":projected?" and a boolean which may be any object but will be evaluated as true or false.

This is equivalent to call with arg list of length 3:

```
(or (sugar t :projected? bool)
    (sugar current-year :projected? bool))
```

The return values are:

1. sugar's fact or yearly value, or *nil* if not known or not obtainable in accordance with the value of ":projected?",
2. *t* if the value is the result of a projection, *nil* otherwise.

Case-3: Assignment of sugar as model-attribute e.g: **(sugar :model t) (sugar :model 1)**.

In this case the arguments are the keyword ":model" and an indicator which may be any object but will be evaluated as either *t* or as something else that is either TRUE or FALSE.

The result of this kind of call makes the attribute a *model attribute* with the indicator acting as follows:

- a *t* indicates that the attribute will be reported
- another TRUE value indicates that it will be a model attribute, but not reported,
- a FALSE value means that it is neither of the previous.

The return values are:

1. sugar's attribute name
2. if inserted as model *t*, otherwise *nil.*

As can be seen, parse-2 operates by delegation and as such is merely a dispatcher.

74a ⟨*parse-2* 74a⟩≡ (99a)

```
(defun parse-2 (att-name arg-lis ht cur-yr &optional (accept-projected? t))
  (let ((first (car arg-lis)))
    (cond
     ((numberp first)
      (parse-2-2-numbers att-name arg-lis ht cur-yr accept-projected?))
     ((eq :model first)
      (parse-2-model-att att-name arg-lis ht cur-yr accept-projected?))
     ((eq :projected? first)
      (parse-2-projected? att-name arg-lis ht cur-yr accept-projected?))
     (t (error "parse-2: Arguments incorrect: ~S~%" arg-lis)))))
```

`parse-2-2-numbers`

This is the helper function for `parse-2` which handles the case of a query for a period of years, i.e. the argument list contains "2 numbers". The function processes by mapping calls to `parse-1` over the number list obtained for *[start-date, stop-date]*. The resulting values are assembled into two lists, the first containing the values returned by the query, the second containing the booleans indicating if the value is projected or not.

Arguments and return values are specified in `parse-2`.

74b ⟨*parse-2-2-numbers* 74b⟩≡ (99a)

```
(defun parse-2-2-numbers (att-name arg-lis ht cur-yr accept-projected?)
  (destructuring-bind
   (start-date stop-date) arg-lis
   ;;(when sgr-trace
   ;;(format t "Parse-2-2-numbers:~%att-name: ~S~%Current year: ~S~%"
   ;;     att-name cur-yr)
   ;;(format t "Start-Date: ~S~%Stop-Date: ~S~%Accept-Projected?: ~S~%"
   ;;     start-date stop-date accept-projected?)
   ;;(break))
   (let ((pair-lis (mapcar #'(lambda (date)
                               (multiple-value-list
                                (parse-1
                                 att-name
                                 (list date)
                                 ht
                                 cur-yr
                                 accept-projected?)))
                           (wut:numlist (wut:abs-year start-date cur-yr)
                                        (wut:abs-year stop-date cur-yr)))))
     (values (mapcar #'car pair-lis)
             (mapcar #'cadr pair-lis)))))
```

`parse-2-model-att`

This is the helper function for `parse-2` which handles the case of assignment of "model attribute" for reporting or not, i.e. the argument list contains ":model *bool*". The function processes by first destructuring the argument list, then validating that the ":model" keyword was indeed bound, and if so makes an internal database call to `lookup` to delegate the work.

Arguments and return values are specified in `parse-2`.

75 ⟨*parse-2-model-att* 75⟩≡ (99a)

```
(defun parse-2-model-att (att-name arg-lis ht cur-yr accept-projected?)
  (declare (ignore cur-yr accept-projected?))
  (destructuring-bind
   (&key model) arg-lis
   ;;(when sgr-trace
   ;;(format t "Parse-2-model-att:~%att-name: ~S~%Current year: ~S~%"
   ;;     att-name cur-yr)
   ;;(format t "model: ~S~%Accept-Projected?: ~S~%"
   ;;     model accept-projected?)
   ;;(break))
   (if (not model) (values att-name nil)
     ;; next line was modified 2006 01 18 to support defreport
     ;; indeed, a TRUE value that is NOT T, means that the attribute
     ;; should be reported!
     (progn (ids:lookup t t ht :val att-name :p? (eq model t))
            (values att-name t)))))
```

`parse-2-projected?`

This is the helper function for `parse-2` which handles the case of assignment of a query filtering on projected values, i.e. the argument list contains ":projected? *bool*". The function processes by first destructuring the argument list, then delegating the work to `parse-0`.

Arguments and return values are specified in `parse-2`.

76 ⟨*parse-2-projected? 76*⟩≡ (99a)

```
(defun parse-2-projected? (att-name arg-lis ht cur-yr accept-projected?)
  "handles the case of :projected? by delegating to parse-0
"
  (destructuring-bind
   (&key projected?) arg-lis
   ;;(when sgr-trace
   ;;(format t "Parse-2-aux:~%att-name: ~S~%Current year: ~S~%"
   ;;     att-name cur-yr)
   ;;(format t "projected?: ~S~%Accept-Projected?: ~S~%"
   ;;     projected? accept-projected?)
   ;;(break))
   (parse-0 att-name arg-lis ht cur-yr (and projected? accept-projected?))))
```

7.3.6 `parse-3`

If argument list is of length 3, there are three cases: one query, and two assignments.

Case-1: Query for a fact or a single year, filtering for "projected?" e.g, **(sugar t :projected? nil) (sugar -1 :projected? t)** In this case the arguments are a date, the keyword ":projected?" and a boolean such that:

date: may be t, an absolute year or relative year,

keyword: must be ":projected?" since all other keywords would entail assignment,

boolean: may be any object but will be evaluated as TRUE or FALSE.

The return values are:

1. the value of sugar for date, accepting projected values if bool is TRUE, and not accepting projections otherwise,
2. *t* if the value is the result of a projection, *nil* otherwise.

Cases-2 and 3: Assignment for a fact or timely value, as projected or not, e.g. **(sugar t :set "john") (sugar 2005 :project 104.5) (sugar 2000 :set 100)** In this case the arguments are a date, either of the keywords ":project" or ":set" and a value to be assigned such that:

date: may be *t*, an absolute year or relative year,

keyword: If keyword is ":project" then the value is assigned and the "projected?" flag is set to TRUE. If keyword is ":set" then the value is assigned and the "projected?" flag is un-set.

value: may be any object.

The return values are:

1. the value that has been projected or set,
2. *t* if the value is projected, *nil* otherwise.

The processing follows the same path as the other parsing functions:

1. After destructuring, selection of the case is determined by the presence or absence of values for the keywords.
2. First, we check on ":set". If this is present then we delegate to the database call with a `lookup` and return the appropriate 2 values,
3. if not, we then check on ":project" and perform the appropriate action similar to the previous case,

4. otherwise, it must be a query on with the "projected?" filter. This is delegated to `parse-1`.

78 ⟨*parse-3* 78⟩≡ (99a)

```
(defun parse-3 (att-name arg-lis ht cur-yr)
  (destructuring-bind
   (date &key set project projected?) arg-lis
   ;;(when sgr-trace
   ;;(format t "Parse-3:~%att-name: ~S~%Current year: ~S~%"  att-name cur-yr)
   ;;(format t "set:: ~S~%project: ~S~%projected?: ~S~%"
   ;;      set project projected?)
   ;;(break))
   (let ((abs-yr (wut:abs-year date cur-yr)))
     (cond
      (set (progn (ids:lookup att-name abs-yr ht :val set :p? nil)
                  (values set nil)))
      (project (progn (ids:lookup att-name abs-yr ht :val project :p? t)
                      (values project t)))
      (t ; then its a query filtering on projected?
       (parse-1 att-name (list abs-yr) ht cur-yr projected?))))))
```

7.3.7 parse-4

If argument list is of length 4, there are two cases: one query, and one assignment.

Case-1: Query on a period while filtering for "projected?" e.g. **(sugar -2 +5 :projected? nil)** In this case the arguments are *start-year, stop-year*, the keyword ":projected?" and a boolean such that:

start-year, stop-year: absolute or relative years,

keyword: must be ":projected?" since all other keywords would entail assignment,

boolean: it may be any object but will be evaluated as TRUE or FALSE.

1. a list of sugar's values for each year on *[start-year, stop-year]*, or *nil* if not known and not projectable,
2. a list with a value for each year that is *t* if the value is the result of a projection, *nil* otherwise.

Case-2: rule assignment, e.g. **(sugar :rule cap-x :prec 20)**. In this case the arguments are keyword-1, value, keyword-2, and a number such that:

keyword-1: is ":rule",

value: may be a function or symbol that evaluates to a function,

keyword-2: is ":prec",

number: is the numerical value of the rule's precedence. This value must be > 0.

1. the function that was assigned as a rule for the attribute,
2. *t* if success, *nil* if not.

This function's processing is delegated to `parse-2-2-numbers` in "case 1" and to `parse-4-rule` in the "case 2".

79 ⟨*parse-4* 79⟩≡ (99a)

```
(defun parse-4 (att-name arg-lis ht cur-yr)
  (if (numberp (car arg-lis))
      (destructuring-bind
       (start-date stop-date &key projected?) arg-lis
       ;;(when sgr-trace
       ;;(format t "Parse-4~%att-name: ~S~%Current year: ~S~%"  att-name cur-yr)
       ;;(format t "Start-Date: ~S~%Stop-Date: ~S~%Projected: ~S~%"
       ;; start-date stop-date projected?)
       ;;(break))
       (parse-2-2-numbers att-name (list start-date stop-date) ht
                          cur-yr projected?))
    (parse-4-rule att-name arg-lis ht cur-yr)))
```

parse-4-rule

This helper function does the rule assignment work for `parse-4`. It does the same destructuring as previously, but with the assurance that both keys will be bound. It delegates the assignment to the database function `lookup`.

80 ⟨*parse-4-rule* 80⟩≡ (99a)

```
(defun parse-4-rule (att-name arg-lis ht unused)
  (declare (ignore unused))
  (destructuring-bind
   (&key rule prec) arg-lis
   ;;(when sgr-trace
   ;;(format t "Parse-4-rule~%att-name: ~S~%"  att-name)
   ;;(format t "Rule: ~S~%Precedence: ~S~%"
   ;;     rule prec)
   ;;(break))
   (ids:lookup att-name () ht :val rule :p? prec)))
```

7.3.8 `find-or-project` (att-name abs-y ht &key (accept-projected? t) (project? t))

This is the main work-horse function which supports the parsing functions. Its job is to first look for data, and then if there is none available, to fire rules until the data is created, or until all available rules have failed.

Arguments:

1. string name of the attribute, or *t* if we are looking for "MODEL-ATTRIBUTES",
2. absolute year value or *t*,
3. the main hash-table,
4. a boolean indicating that projected values are or are not acceptable,
5. a boolean indicating that it is ok to project or not.

Return: In the case of an attribute of type string:

1. the value of the attribute for the abs-year or *nil* if not found, and not projectable depending on the booleans and the result of rule firings,
2. *t* if the value is the result of a projection, *nil* if not projected or if no value was found,

In the case of the att-name *t*:

1. list of model-attribute names,
2. *t*.

This function processes as follows:

1. First, a `lookup` finds the value, if present in the database,
2. then, if the attribute was *t*, and results were found, then the results are formatted with a mapcar, and returned,
3. if the attribute is not *t*, but data was found, we filter the return for "projected" values as needed with a call to `return-val-projected?`,
4. if no value is found, and we are allowed to project, then we delegate the return to `project`,
5. finally, there's nothing left but to give up and return the failure response *nil; nil*.

82 ⟨*find-or-project* 82⟩≡ (99b)

```
(defun find-or-project (att-name abs-y ht
                                &key (accept-projected? t) (project? t))
  (multiple-value-bind
   (val found?) (ids:lookup att-name abs-y ht)
   ;;(when sgr-trace
   ;;(format t "Find-or-Project:~%att-name: ~S~%abs-y: ~S~%"  att-name abs-y)
   ;;(format t "Accept-projected: ~S~%project?: ~S~%"
   ;;     accept-projected? project?)
   ;;(format t "Val: ~S~%Found: ~S~%"
   ;;     val found?)
   ;;(break))
   (cond
    ((and (eq t att-name) found?) (values (mapcar #'car val) t))
    (found? (return-val-projected? val accept-projected?))
    ((and accept-projected?
          project?)  (project att-name abs-y ht))
    (t (values ()())))))
```

7.3.9 `project` (att-name abs-date ht)

This function is called when it is necessary to project data in order to reply to a query. It works by finding the applicable rules for the attribute in question, firing them in the correct order, and returning the result of the first successful firing, or signalling a failure if none succeeded.

Arguments:

1. an attribute name,
2. an absolute date,
3. the main hash-table.

Return:

1. the projected value, or *nil* if projection fails,
2. *t* if projected with success, *nil* in all other cases.

Using `some-rule`, we map over all the applicable rules. If a value is produced, then it is assigned in a call to `lookup` and returned with the projected indicator *t*. Otherwise, the failure response, *nil; nil,* is returned.

83 ⟨*project* 83⟩≡ (99b)

```
(defun project (att-name abs-date ht)
  (let ((value (some-rule (ids:get-rules att-name :ht ht) att-name abs-date)))
    ;;(when sgr-trace
    ;;(format t "Project:~%att-name: ~S~%abs-yr: ~S~%"  att-name abs-date)
    ;;(format t "Value: ~S~%"
    ;;      value)
    ;;(break))
    (if value
        (progn (ids:lookup att-name abs-date ht :val value :p? t)
               (values value t))
      (values ()()))))
```

7.3.10 `some-rule` (rule-func-lis att-name abs-date)

This is a mapping function, similar to the Common Lisp function *some*. It applies all the rules in its first argument to the arguments *(att-name abs-date)* and returns the first non *nil* result, or *nil* if none return a TRUE value.

Arguments:

1. a list of rule-functions in a form callable by "apply" or "funcall",
2. a string attribute name,
3. an absolute date.

Return:

1. The value that was produced by the 1st successful rule execution or *nil* if all failed,
2. The second value of the rule function application, or *nil* if no functions return a value.

The processing is recursive and proceeds as per:

1. First, if there are no more rules in the rule-func-lis, then we return the failure values,
2. if there are elements in the rule-func-lis, then first send the rule-func to `loop-detect`,
3. then use a *funcall* to execute the selected rule-func,
4. then when the rule-func returns, remove it from the loop detector,
5. and, if there was a valid value returned, this is the return,
6. otherwise recurse on the remaining elements in rule-func-lis.

84 ⟨*some-rule* 84⟩≡ (99b)

```
(defun some-rule (rule-func-lis att-name abs-date)
  (if(null rule-func-lis) (values () ())
    (let ((rule-func (car rule-func-lis)))
      (loop-detect +1 rule-func att-name abs-date)
      (multiple-value-bind
       (val-1 val-2)
       (funcall rule-func att-name abs-date)
       (loop-detect -1)
       (if val-1 (values val-1 val-2)
         (some-rule (cdr rule-func-lis) att-name abs-date))))))
```

7.3.11 `return-val-projected?` (val-pair accept-projected?)

This is a conversion helper function used to filter a pair of the form `(val . bool)` for projected? or not.

Arguments:

1. a pair `(value . t)` or `(value . nil)` where the cdr indicates projected or not,
2. a boolean indicating if projected values are acceptable or not.

Return:

1. value, or *nil* if projected conflicts with "accept-projected?",
2. the value of "projected?".

85 ⟨*return-val-projected?* 85⟩≡ (99b)

```
(defun return-val-projected? (val-pair accept-projected?)
  (let ((v (car val-pair))
        (p (cdr val-pair)))
    ;;(when sgr-trace
    ;;(format t "Return-val-Projected?:~%val-pair: ~S~%"  val-pair)
    ;;(format t "Accept-projected?: ~S~%"
    ;;       accept-projected?)
    ;;(break))
    (if (or (not p)
            accept-projected?)
        (values v p)
      (values () ()))))
```

7.4 How are Sugar Functions Stored?

We must now look at the operation of the SUGAR package itself. There are quite a few things to look at here. The SUGAR package must be cleared out at each load of a new data set, since if not, the sugar functions would point to the wrong data. Similarly, the special *sugar functions* must be regenerated for each new data set. Finally, the reset and loop detection mechanisms must also be reset.

All of these operations are performed by the following function.

7.4.1 `init-sugar` (ticker current-year &key loop-detect)

This is the start of any data set load.

Arguments:

1. string name for the ticker naming the data set,
2. the current year as number
3. a boolean which if TRUE activates loop detection, otherwise loop detection is disabled (the default).

Return:

- the internal database.

Processing proceeds as per:

1. First, create the internal database,
2. then, nullify any existing *sugar functions*,
3. then create *sugar functions* for *t* and for *nil*, called “MODEL-ATTRIBUTES” and “GENERAL-RULES”.
4. then, create a *sugar function* for “CURRENT-YEAR” and set it’s fact value.
5. create a *sugar function* for “TICKER” and set it’s fact value.
6. create the special *rule function* “simple-get-data,”
7. create the “sugar-nullifier” and the “loop-detector”,
8. finally, return the internal database.

86 ⟨*init-sugar* 86⟩≡ (98e)

```
(defun init-sugar (ticker current-year &key loop-detect)
  (let* ((db (ids:make-db 0)))
    (nullify-sugar)
    (create-simple-get-data db)
    (create-sugar-function t db current-year)
    (create-sugar-function () db current-year)
    (funcall
     (att-name-2-sugar-func "CURRENT YEAR"  db current-year)
     t :set current-year)
    (funcall
     (att-name-2-sugar-func "TICKER"  db current-year)
     t :set ticker)
    (create-sugar-function-nullifier db)
    (create-loop-detector loop-detect)
    db))
```

7.5 Where do Sugar Functions Go when No Longer Needed?

Are you wondering what this section could possibly be about? If you are, then you're in the right place. If not, well, just fake it and read on.

Sugar functions are closures and as such contain pointers to data. This data is enclosed at the time when the *sugar function* is created. Amongst this data is a pointer to the internal database containing all the data for the current data load. When a new set of data is loaded, a new internal database is created and any *sugar functions* must be set to point to it. The easiest way to do this is to simply "un-defun" the old *sugar functions* and create fresh ones. This is the strategy which has been implemented and is called "nullification".

Nullification is done in two phases, first the the function which will do the nullification is created. It is called, `nullify-sugar`, and is created by a call to `create-sugar-function-nullifier`. Later on, when `nullify-sugar` is called, the actual work of nullifying is delegated to `nullify-sugar-functions`.

7.5.1 `create-sugar-function-nullifier` (db)

This tiny function is used to "create a function" that when called will nullify all the *sugar functions* that are known. This is done by creating a closure object encapsulating the db argument and setting function value of the symbol "nullify-sugar" to that closure.

Argument:

1. an internal database.

Return:

- the *#'lambda* closure encapsulating the call to `nullify-sugar-functions`.

88a ⟨*create-sugar-function-nullifier* 88a⟩≡ (98e)
```
(defun create-sugar-function-nullifier (db)
  (setf (symbol-function 'nullify-sugar)
        #'(lambda()
            (nullify-sugar-functions db))))
```

`nullify-sugar` ()

This is just a placeholder definition needed by the Common Lisp reader for the definition of `create-sugar-function-nullifier` during the initial loading of the SUGAR package.

88b ⟨*nullify-sugar* 88b⟩≡ (98e)
```
(defun nullify-sugar ()
  "just a placeholder for the function which is really
defined in create-sugar-function-nullifier")
```

7.5.2 `nullify-sugar-functions (db)`

This is the function that is called by `nullify-sugar` when it is time to nullify all the known *sugar functions*.

It proceeds by setting all the known sugar functions to the null-sugar-function. It works by mapping, over all the attributes in the system, a *lambda* that sets the attribute's symbol-function to the null *sugar function* returned by the call to `make-null-sugar-func`

Arguments:

1. an internal database.

Return:

- list of attributes whose sugar functions have been nullified.

89 ⟨*nullify-sugar-functions* 89⟩≡ (98e)

```
(defun nullify-sugar-functions (db)
  (mapc #'(lambda(att-name)
            (let ((sugar-func-symbol (sugar-function-symbol att-name)))
              (setf (symbol-function sugar-func-symbol)
                    (make-null-sugar-func sugar-func-symbol))))
        (union (ids:get-keys db)
               (ids:get-model-attributes :ht db)
               :test #'string=)))
```

7.5.3 `make-null-sugar-func` (name)

This is the final step in the nullification process. This function returns a "null" *sugar function* which if ever called would mean that somehow a deprecated *sugar function* has been called, i.e. something would be very wrong. Needless to say, the function simply raises an error and aborts processing. It sends a message to std-err that explicitly indicates which *sugar function* was improperly called.

Argument:

1. a *sugar function* name as a symbol.

Return:

- the null sugar function as a lambda expression.

90 ⟨*make-null-sugar-func* 90⟩≡ (98e)

```
(defun make-null-sugar-func (name)
  (let ((message-string
         (concatenate 'string
                      "Call to inexistent sugar-function!!~%"
                      "The function was called as: ~A")))
    #'(lambda(&rest r)
        (error  message-string
                (append
                 (list
                  (concatenate
                   'string
                   "SGR:"
                   (symbol-name name))) r)))))
```

7.6 Attribute Name "Apply" function and Helpers

7.6.1 `apply-attribute-sugar` (att &rest args)

This function is similar to the Common Lisp function "funcall". It takes an attribute name as a string and applies that attribute's *sugar function* to the rest of the arguments. Oops, we probably should have called it "func-attribute-sugar" but then what would people think? In the end, no one's perfect. We'll just have to continue to live with this slight misnomer . . . .

Arguments:

1. a string name of an attribute,
2. all possible valid arguments to a *sugar function*.

Return:

- The result of the application of the *sugar function* corresponding to the first argument, to the rest of the arguments. An error is raised if there is no corresponding *sugar function*.

The processing is straightforward:

1. first, get the symbol corresponding to the attribute string,
2. then, find the symbol's function value,
3. if there's no function value, signal the error,
4. finally, return the value of the application of the symbol-function to the arguments.

There is no creation of *sugar function*; if none exists an error will be raised.

91 ⟨*apply-attribute-sugar* 91⟩≡ (98c)

```
(defun apply-attribute-sugar (att &rest args)
  (let* ((sym (sugar-function-symbol att))
         (att-sugar-func
          (symbol-function
           (if sym sym
             (error
              "apply-attribute-sugar:  Attribute has no sugar function: ~S~%"
              att)))))
    ;;(when sgr-trace
    ;; (format t "apply-attribute-sugar: ~S~%" att)
    ;;(break))
    (apply att-sugar-func args)))
```

7.6.2 remove-double-spaces (string)

This helper function cleans up attribute string names so that they can be used for the generation of the corresponding symbol name. It removes multiple spaces as well as leading and trailing space. It can handle empty strings.

Argument:

1. A string, can be empty.

Return:

- The string with no leading or trailing spaces, and with all multiple spaces replaced by a single space.

92 ⟨*remove-double-spaces* 92⟩≡ (98c)

```
(defun remove-double-spaces (string)
  (string-trim
   " "
   (reduce #'(lambda(l r)
               (let ((last-char (char l (1- (length l))))
                     (next-char (char r (1- (length r)))))
                 (if (char= #\Space last-char next-char)
                     l
                   (concatenate 'string
                                l
                                r))))
           (map 'list #'(lambda(char)
                          (format nil "~A" char))
                string)
           :initial-value " ")))
```

7.7 Loop detection

What is this? More stuff? Shouldn't this file be broken up into smaller ones? Who wrote this anyway?

Well, we're sorry to admit that all those remarks are perfectly justified and someone should do something about it ...

SAL provides a *Loop Detection* mechanism to help the end user to debug rule functions. When could a rule loop occur? Well suppose there is a query for data "A" and a rule is fired launching a query for data "B". This in turn may fire a rule which launches a query for data "A" again, and we have a loop. SAL can detect this and inform the user as to what went wrong.

7.7.1 `create-loop-detector` (activate)

The first part of loop detection is the creation of the loop detector function. The loop detector function, called `loop-detect` is called at every rule firing and at every return from a rule firing. At the rule firing a counter is incremented, and at the return the counter is decremented. Also, at each firing the rule's arguments are recorded and compared with all those previously recorded. In this manner, if a rule is called twice with the same set of arguments, it will be detected and the user can be informed.

Arguments:

1. boolean, if TRUE activate, otherwise deactivate loop detection.

Return:

- The *#'lambda* closure encapsulating the rule-firings list.

loop detector: lambda (inc &rest args)

This is the function that is actually called before and after each rule firing (if loop detection is enabled).

Arguments:

1. a number: if positive then the firing is pushed onto the internal list of firings; if negative, we are returning from a firing so the last firing is popped from the internal list,

2. rest of args, i.e. a tuple: `(#'rule-func attribute yr)`.

Return:

- if a loop is detected, exit by error with display of the loop delegated to `show-loop`, otherwise the updated list of firings tuples is returned.

The lambda takes an argument $+1$ if adding a firing, -1 if returning from a firing and checks the args for having already been seen in a previous call. If so, error, if not the args are pushed/popped form the list of firings and execution continues.

94a ⟨*create-loop-detector* 94a⟩≡ (98d)

```
(defun create-loop-detector (activate)
  (if activate
      (let ((firings-lis ()))
        (setf (symbol-function 'loop-detect)
              #'(lambda (inc &rest args)
                  (if (minusp inc) (pop firings-lis)
                    (if (member args firings-lis :test #'equal)
                        (error "Loop-Detector: rule loop detected:~%~:@W"
                               (show-loop args firings-lis))
                      (push args firings-lis))))))
    (setf (symbol-function 'loop-detect)
              #'(lambda (inc &rest args)
                 (declare (ignore inc args))))))
```

`loop-detect` (inc &rest args)

This is just a placeholder definition needed by the Common Lisp reader for the definition of `create-loop-detector` during the initial loading of the SUGAR package.

94b ⟨*loop-detect* 94b⟩≡ (98d)

```
(defun loop-detect (inc &rest args)
  (declare (ignore inc args)))
```

7.7.2 `show-loop` (tuple tuple-lis)

This function is used to display the detected loop to the user via the call to "error". It returns a string which contains the rule firings, with their arguments, in the sequence which has looped.

Arguments:

1. the tuple that caused the loop,
2. the firing tuples in the order of last-in, first-out.

Return:

- a string showing the sequence of firings.

95a ⟨*show-loop* 95a⟩≡ (98d)

```
(defun show-loop (tuple tuple-lis)
  (let* ((full-lis (append (reverse tuple-lis) (list tuple)))
         (first (car full-lis)))
    (reduce #'(lambda(l-tuple r-tuple)
                (format nil "~A~%~S" l-tuple r-tuple))
            (cdr full-lis) :initial-value (format nil "~S" first))))
```

7.8 SAL's private Sugar Functions

7.8.1 `create-simple-get-data` (ht)

This function will create and assign a special *rule function* for the internal database key-pair *(nil, nil)*, which will call `find-or-project` on the pair *(attribute, date)* with ":project?" set to FALSE. In other words, executing this function creates a rule function of precedence *zero,* i.e. the first rule function to be called in all cases, which is general for all attributes, and which will simply look up its value in the internal database. It will never project.

Argument:

- an internal database hash-table.

Return:

1. the value found, or *nil* if unknown,
2. a boolean TRUE indicating that the value is projected, or FALSE if not projected or if absent.

NOTE: The rule function created by this call will NOT project data.

95b ⟨*create-simple-get-data* 95b⟩≡ (98e)

```
(defun create-simple-get-data (ht)
  (ids:lookup () () ht
              :val #'(lambda(at yr)
                       (find-or-project at yr ht :project? nil))
              :p? 0))
```

7.8.2 `industry (&rest r)`

This is a dummy function definition, used as a placeholder until the true "industry" *sugar function* becomes available. It is needed prior to package initialization, just so that the symbol "industry" is defined.

96 ⟨*industry* 96⟩≡ (98e)
```
(defun industry (&rest r)
  (declare (ignore r))
  (values ()()))
```

7.9 Test Harness and Debugging Helpers

97 ⟨*sugar-test-harness* 97⟩≡ (98b)

```
(defparameter *c-c-alis* ())
(setf *c-c-alis*
  '(("Init Sugar and create a Stock-db called 'd':"
     .
     "(format t \"~S~%\" (setf d (sgr:init-sugar \"ibm\" 2005)))")
    ("Create a sugar func for \"toto\":"
     .
     "(format t \"~S~%\" (sgr:att-name-2-sugar-func \"toto\" d ))")

    ("Insert the 4-tuple: \"toto\" 1995 5 NOT-projected!"
     .
     "(format t \"~S~%\" (multiple-value-list (sgr:toto 1995 :set 5)))")
    ("Insert the 4-tuple: \"toto\" 1996 6 PROJECTED!"
     .
     "(format t \"~S~%\" (multiple-value-list (sgr:toto 1996 :project 6)))")
    ("Get the value of \"toto\" in 1997"
     .
     "(format t \"~S~%\" (multiple-value-list (sgr:toto 1997 )))")
    ("Get the values of \"toto\" on [1994 2000]"
     .
     "(format t \"~S~%\" (multiple-value-list (sgr:toto 1994 2000)))")
    ("Get only the database values of \"toto\" on [1994 1998]"
     .
     "(format t \"~S~%\" (multiple-value-list (sgr:toto 1994 1998 :projected? nil)))")
    ("Examine the Stock-db called 'd':"
     .
     "(format t \"~S~%\" d)")
    ("Create a sugar func for \"ceo\":"
     .
     "(format t \"~S~%\" (sgr:att-name-2-sugar-func \"ceo\" d ))")
    ("Insert the tuple: \"ceo\" t \"John\""
     .
     "(format t \"~S~%\" (multiple-value-list (sgr:ceo t :set \"John\")))")
    ("Who is the ceo?"
     .
     "(format t \"~S~%\" (multiple-value-list (sgr:ceo)))")
    ("Who is the ceo stupidly asked for 2010?"
     .
     "(format t \"~S~%\" (multiple-value-list (sgr:ceo 2010)))")
    ("Who is the ceo stupidly asked for [2000 2005]?"
     .
     "(format t \"~S~%\" (multiple-value-list (sgr:ceo 2000 2005)))")
    ))
```

98a ⟨*sugar-debugging-helpers* 98a⟩≡ (98b)
```
;;(defparameter sgr-trace ())
```

7.10 Physical Layout of the File

The package is ordered as per the following:

98b ⟨*sugar.lisp* 98b⟩≡
```
;;; sugar.lisp
```
⟨*lisp-header* 143b⟩
⟨*sugar-package-def* 61⟩
⟨*sugar-debugging-helpers* 98a⟩
⟨*attribute-sugar-function-manipulators* 98c⟩
⟨*loop-detection* 98d⟩
⟨*sugar-package-init* 98e⟩
⟨*parsing-functions* 99a⟩
⟨*parsing-helpers* 99b⟩
⟨*sugar-execution* 99c⟩
⟨*eoc* 143c⟩
⟨*sugar-test-harness* 97⟩
⟨*eof* 144⟩

Each of the subsections of the file is defined by the following sets of functions:

98c ⟨*attribute-sugar-function-manipulators* 98c⟩≡ (98b)
⟨*remove-double-spaces* 92⟩
⟨*sugar-function-name* 65⟩
⟨*sugar-function-symbol* 64b⟩
⟨*attribute-sugar-function* 64a⟩
⟨*apply-attribute-sugar* 91⟩
⟨*att-name-2-sugar-func* 63⟩

98d ⟨*loop-detection* 98d⟩≡ (98b)
⟨*show-loop* 95a⟩
⟨*loop-detect* 94b⟩
⟨*create-loop-detector* 94a⟩

98e ⟨*sugar-package-init* 98e⟩≡ (98b)
⟨*create-simple-get-data* 95b⟩
⟨*create-sugar-function* 66⟩
⟨*make-null-sugar-func* 90⟩
⟨*nullify-sugar-functions* 89⟩
⟨*nullify-sugar* 88b⟩
⟨*create-sugar-function-nullifier* 88a⟩
⟨*industry* 96⟩
⟨*init-sugar* 86⟩

99a ⟨*parsing-functions* 99a⟩≡ (98b)
⟨*parse-0* 70⟩
⟨*parse-1* 72⟩
⟨*parse-2* 74a⟩
⟨*parse-2-2-numbers* 74b⟩
⟨*parse-2-model-att* 75⟩
⟨*parse-2-projected?* 76⟩
⟨*parse-3* 78⟩
⟨*parse-4* 79⟩
⟨*parse-4-rule* 80⟩

99b ⟨*parsing-helpers* 99b⟩≡ (98b)
⟨*find-or-project* 82⟩
⟨*project* 83⟩
⟨*some-rule* 84⟩
⟨*return-val-projected?* 85⟩

99c ⟨*sugar-execution* 99c⟩≡ (98b)
⟨*parse-funcs* 68b⟩
⟨*exec-sugar* 68a⟩

7.11 SUGAR Package History

2005 12 17: GF creation of the file.

2005 12 19: GF created some new functions and adapted so as to be able to generate sugar for "MODEL-ATTRIBUTES" and "GENERAL-RULES".

2005 12 28: GF update to handle package-defs.lisp and correct name-space problems.

2006 01 11: GF update to move `sugar-function-symbol` from rule-funcs.lisp, create new function: `apply-attribute-sugar`

2006 01 18: GF modify `parse-2-model` to handle use of model list as report spec. this means using a significant value instead of simply t and nil as value of keyword :model in sugar call `(toto :model 1)`.

2006 01 29: GF update to create sugar-func-lis and associated mgt.

2006 02 01: GF made the test harness `*C-C-ALIS*` private to the SGR package.

2006 02 03: GF cleaned up the exported symbol list, created the function: `attribute-sugar-function`, repaired defect in `att-name-2-sugar-func`!!! Tested OK!

2006 02 05: GF update and repair error in att-name-2-sugar-func to test also model attributes before sugar-function creation. Remove *sugar-function-lis* and use a closure on nullify-sugar to hand clean-up of sugar-functions on init. Update as per code-review, too! Fully tested, all is 100% + performance improvement of 5%!!!

2006 02 06: GF update to implement loop detection in rule-firings!

2006 02 07: GF update to complete loop detection in rule-firings! comment out all tracing

2006 02 09: GF update init-sugar to uppercase "CURRENT YEAR" and "TICKER" remove all reference to stock-data-structure which has become obsolete. This wide-ranging change has been tested and seems ok, although there seems to be a slight increase in db loading time.

2006 02 10: GF create placeholder industry function. Fixed defect in create sugar function! Optimization in show-loop, following Blake's remark!

2006 02 19: GF moved `sugar-function-name` and `remove-double-spaces` to this file from utilities.lisp.

2006 04 30: GF corrected bug in `parse-1` which prevented projection with $date = T$.

2006 05 17: GF improved ordering of functions during conversion to Literate style.

2006 05 22: GF simplified the logic of `parse-3` to better reflect the 3 cases described.

Chapter 8

Rule Support Functions

The application of rules to compute missing data and to project it into the database is also at the heart of SAL. Indeed, SAL is a backward chaining rule-based deduction engine and as such support for the manipulation of rules is *the* fundamental element of SAL seen by the user.

The RULE-FUNCS package provides that support.

8.1 The RULE-FUNCS Package

The package definition shows the five exported symbols. These will be detailed in the order of importance and in a depth first exploration.

102 ⟨*rule-funcs-pkg-def* 102⟩≡ (110a)

```
(defpackage "RULE-FUNCS"
  (:use "COMMON-LISP" "SUGAR")
  (:nicknames "RF")
  (:export "INIT"
           "DEFRULE"
           "DEFREPORT"
           "DEFMODEL"
           "DEFDATA"
           ))

(in-package "RULE-FUNCS")
```

8.1.1 init (db)

Package initialization is performed by this function. This comprises the assignment of a pointer to the SAL internal database to the package local variable `*sal-db*`.

Arguments:

1. an internal database.

Return:

- the same value as the input argument.

103a ⟨*rf:init* 103a⟩≡ (110a)

```
(defun init (db)
  ;;(when rf-trace
  ;;(format t "rf:init: ~A" db)
  ;;(break))
  (setf *sal-db* db))
```

`*sal-db*`

This is a package local variable that must be set to point to an internal database before any of the `RULE-FUNCS` package functionality is accessed.

103b ⟨**sal-db** 103b⟩≡ (110a)

```
(defparameter *sal-db* ())
```

8.1.2 defrule (<args>)

This macro defines a rule, and loads it into the memory database.

The full profile of the `defrule` macro is:

```
(defmacro defrule (rule-name
                   applicable-att-lis
                   precedence
                   rule-arguments
                   &rest body)
```

Arguments:

1. the name of the rule as an un-quoted symbol,
2. list of attributes for which it should be applied, these are strings, except for *t* which is used to indicate a general rule,
3. a number indicating precedence,
4. a list of formal arguments that will be the ones referenced in the rule's body. This list must be of length 2. The formal arguments should usually be called 'att' and 'yr', since the rule-function will be called with an attribute and a year as arguments,
5. the rest is the body of the rule function.

Return:

- The rule's definition as a lambda.

Example Calls:

```
(rf:defrule general (t) 5 (unused-1 unused-2)
  (format t "General rule!~%")
  (format t "the end."))

(rf:defrule  example-printer-rule ("Cash flow" "Profit") 20 (att yr)
  (format t "Example Printer Rule called:~%Attribute: ~S~%Year: ~S~%" att yr)
  (format t "The end."))
```

The processing of a `defrule` call is as follows:

1. First, note that `defrule` is a *macro*, which means that it's arguments are not evaluated at the time of the call.
2. Processing begins by ensuring the the *precedence* is > 0. If it is not, then an error is raised and processing aborts.
3. if the the precedence is valid, then a local variable *f-def* is initialized with the rule-function's definition built from the arguments,

4. next, the call is logged,
5. then, the function value of the rule-name symbol is set to the previously constructed function definition,
6. then the symbol is made accessible from the RULE-FUNC package,
7. finally, a *sugar function* call is used in a mapping over all the applicable attributes, associating the rule-function to the attribute at the specified precedence,
8. at the end the function definition is returned for good measure.

Note: Care should be made in case of multiple calls with same rule-name without undefining the rule, since the system will only maintain the first association that has been entered.

105 ⟨*defrule* 105⟩≡ (110a)

```
(defmacro defrule (rule-name
                   applicable-att-lis
                   precedence
                   rule-arguments
                   &rest body)
  (if (> precedence 0)
      (let ((f-def `(lambda ,rule-arguments ,@body)))
        (wut:log-it (format nil "defrule: ~S" rule-name))
        (setf (symbol-function rule-name) (eval f-def))
        (import rule-name "RULE-FUNCS")
        (export rule-name "RULE-FUNCS")
        (mapcar #'(lambda(target-att)
                    (apply-attribute-sugar
                     (if (eq target-att t)
                         ()
                       target-att)
                     :rule rule-name
                     :prec precedence))
                applicable-att-lis)
        ;;(when rf-trace
        ;;(format t "Definition of rule: ~A~%" rule-name)
        ;;(format t "~:@W~%" f-def)
        ;;(break))
        f-def)
    (error (concatenate
            'string
            "Defrule: precedence must be greater than zero!~%"
            "Rule: ~S~%Precedence: ~S~%")
           rule-name precedence)))
```

8.1.3 `defreport` (&rest att-lis)

This function takes an unspecified number of attributes, in the form of strings, and sets them to be reported by SAL. The actual work is performed by the helper function `model-report-helper`.

Arguments:

1. no specific number of arguments, all are assembled into a list for processing by a helper function. The arguments should be strings.

Return:

- list of strings that were provided as arguments.

106a ⟨*defreport* 106a⟩≡ (110a)

```
(defun defreport (&rest att-lis)
  (model-report-helper att-lis t))
```

8.1.4 `model-report-helper` (att-lis report?)

This helper function does the work of both `defreport` and `defmodel`.

The report? argument is *t* when helping a `defreport` call, and any other TRUE value when helping `defmodel`. In all cases the return value is the same. This simply applies the attribute's *sugar function* with the appropriate arguments making the attribute a *model-attribute* with or without reporting.

Arguments:

1. the list of attributes (as strings) to be declared model or reportable,
2. a boolean: if eq to *t* indicating that the attributes are to be reported, any other TRUE value means that the attribute is a model attribute not to be reported.

Return:

- The list of attributes.

106b ⟨*model-report-helper* 106b⟩≡ (110a)

```
(defun model-report-helper (att-lis report?)
  (mapc #'(lambda(att)
            ;;(when rf-trace
            ;;(format t "model-att-helper: att: ~S   report:  ~S~%"
            ;;      att report?)
            ;;(break))
            (funcall (sgr:att-name-2-sugar-func att
                                                 *sal-db*)
                     :model report?))
        att-lis))
```

8.1.5 `defmodel` (&rest att-lis)

This function takes an unspecified number of attributes, in the form of strings, and sets them to be *model-attributes* NOT reported by SAL. The actual work is performed by the helper function `model-report-helper`.

Arguments:

1. no specific number of arguments, all are assembled into a list for processing by a helper function. The arguments should be strings.

Return:

- list of strings that were provided as arguments.

107a ⟨*defmodel* 107a⟩≡ (110a)

```
(defun defmodel (&rest att-lis)
  ;;(when rf-trace
  ;;(format t "Defmodel: ~S~%" att-lis)
  ;;(break))
  (model-report-helper att-lis 1))
```

8.1.6 `defdata` (&rest data-lis)

This function is used to define data in SAL's internal database. It takes any tree-structure in which the leaves are tuples of the form `("toto" val date)` or `("titi" val)`, and loads them into the database. The function reports to std-out all the data that has been rejected (there are some tests to ensure that the data is well formed as well as the total number of loaded data. The actual work is done by the helper function `defdata-helper`.

Arguments:

1. a tree with leave tuples of the form `(att val optional-year)`.

Return:

- the number of data loaded, which could be zero if none were loaded.

Note: the *val*must be *non-nil* otherwise it will be rejected.

107b ⟨*defdata* 107b⟩≡ (110a)

```
(defun defdata (&rest data-lis)
  ;;(when rf-trace
  ;;(format t "Defdata: ~S~%" data-lis))
  (let ((nb (defdata-helper data-lis)))
    (wut:log-it (format nil "Data loaded: ~S" nb))
    nb))
```

8.1.7 `defdata-helper` (data-tree)

This function recursively parses the tree of data provided by `defdata`. On the way down the tree it explores the branches and when it finds a leaf, it delegates to `load-datum` the work of loading the element into SAL's internal database.

On the returns from the recursion, a number of elements loaded is provided. These numbers are summed by means of an embedded call to *reduce* and #'+.

Arguments:

1. a tree with leaves of the form `(att val optional-year)`, or a number in the case of a recursive call.

Return:

- the number of data loaded.

This function processes using a selection on the argument:

1. If the argument is an empty list, then we have reached the end of the tree, just return the number *zero*,
2. if the argument is a number, then we are in a recursive return, just pass the number along,
3. if we have a list of lists, then we must handle two cases:
   (a) if the list is of length 1, then we cannot use *reduce*, so we simply continue the tree exploration on the single element of the list,
   (b) otherwise, the list has more than one element, so we explore both the *car* and the *cdr* and sum the returns by means of a *reduce* call with an embedded +.
4. if the argument is a simple list, then it must be data. Delegate the loading to `load-datum` which should return the number of data loaded.

108 ⟨*defdata-helper* 108⟩≡ (110a)

```
(defun defdata-helper (data-tree)
  ;;(when rf-trace
  ;; (format t "defdata-helper: ~S~%" data-tree)
  ;; (break))
  (cond
   ;; nothing to do, return ZERO
   ((null data-tree) 0)
   ;; if it's a number, we're recursing, just return it to be counted.
   ((numberp data-tree) data-tree)
   ;; a special case since we can't run #'reduce on a list of length 1.
   ((and (listp (car data-tree))
         (= 1 (length data-tree)))
    (defdata-helper (car data-tree)))
   ;; data-tree is list of lists, map & reduce
   ((listp (car data-tree)) (reduce #'(lambda(l r)
```

```
                                                ;;(when rf-trace
                                                ;;(format t
                                                ;;  "lambda~%l
                                                ;;        ~S~%r ~S~%" l r)
                                                ;;(break))
                                                (+ (defdata-helper l)
                                                   (defdata-helper r)))
                                            data-tree))
    ;; ok, it's a leaf-node, i.e. a datum!
    (t (load-datum data-tree))))
```

8.1.8 `load-datum` (tuple)

This function is called to load a tuple of the form `(att val optional-year)` into SAL's internal database. It uses the attribute's *sugar function* to do the loading.

- If length is 2 then we have a fact,
- Otherwise, we have temporal data.
- If the value is *nil*, we reject it.

Arguments:

1. a tuple: `(att val optional-year)`.

Return:

- 1 if the datum was successfully loaded, 0 otherwise.

109 ⟨*load-datum* 109⟩≡ (110a)

```
(defun load-datum (tuple)
 (let ((att  (nth 0 tuple))
       (val  (nth 1 tuple))
       (date (nth 2 tuple)))
   ;;(when rf-trace
   ;;(format t "Load-Datum: ~S~%" tuple)
   ;;(break))
   (cond
    ;; is it valid?
    ((null val) (wut:log-it
                 (format nil "Nil datum rejected: ~S" tuple)) 0)
    ;; use the sugar, but check if it is a fact or a temporal-datum
    (t (funcall (sgr:att-name-2-sugar-func att
                                          *sal-db*)
                (if date date t) :set val)
        1))))
```

8.2 Physical Layout of the File

The package is ordered as per the following:

110a ⟨*rule-funcs.lisp* 110a⟩≡

```
;;; rule-funcs.lisp
⟨lisp-header 143b⟩
⟨rule-funcs-pkg-def 102⟩
⟨rule-funcs-debugging-helpers 110b⟩
⟨*sal-db* 103b⟩
⟨rf:init 103a⟩
⟨defrule 105⟩
⟨model-report-helper 106b⟩
⟨defreport 106a⟩
⟨defmodel 107a⟩
⟨load-datum 109⟩
⟨defdata-helper 108⟩
⟨defdata 107b⟩
⟨eof 144⟩
```

Debugging Helpers

110b ⟨*rule-funcs-debugging-helpers* 110b⟩≡ (110a)

```
;;(defparameter rf-trace ())
```

8.3 RULE-FUNCS Package History

2005 12 17: GF creation of the file.

2005 12 18: GF rewrite in new paradigm

2005 12 29: GF update to include rule-prec list and report-att-lis.

2005 12 30: GF update to added use-package "SUGAR" to eliminate "sgr:" prefixes.

2006 01 11: GF update to move sugar-function-symbol to sugar.lisp, change use-previous to use newly available function `apply-attribute-sugar`.

2006 01 15: GF update to implement `defrule`.

2006 01 18: GF update to implement `defrule`, `defreport`

2006 01 22: GF update to implement `defdata`,

2006 01 26: GF update to create an init, and db parameter for structural and dependency reasons.

2006 02 03: GF minor updates after code-review; update defrule to prevent rules with precedence == 0.

2006 02 07: GF commented out all tracing.

2006 02 09: GF removed references do dbs, as stock-data-structure is now obsolete.

Chapter 9

Internal Data Structures

This is the description of SAL's internal data structures.

SAL uses an in-memory database to store associations of the form *attribute* $\rightarrow$ *value* and $(attribute, year) \rightarrow value$. The former are known as *factual* and the latter are called *temporal* or *timely* values.

This database is implemented as a hash-table whose *key* $\rightarrow$ *value* associations are specified as follows:

Primary Hash-Table

t $\rightarrow$ a list of **model-attribute pairs**,

nil $\rightarrow$ a list of *universal* **rule tuples**, meaning rule tuples which refer to rules that are associated with all attributes. This list is ordered according to increasing precedence values,

a string $\rightarrow$ a **secondary hash-table**.

Model-Attribute Pair

car a *model attribute* as a string,

cdr either *t* indicating that the *model attribute* should be reported, or *nil* if not to be reported.

Rule Tuple

car a *rule function*,

cdr a numerical precedence value.

Secondary Hash-Table

t → a **value pair** corresponding to the attribute's (string key from primary hash-table) value as a fact,

nil → a list of *specific* **rule tuples**, meaning rule tuples which refer to rules that are associated with only this attribute. This list is ordered according to increasing precedence values,

a year → a **value pair** corresponding the attribute's (string key from primary hash-table) value for year (as a number).

Value Pair

car any lisp object, the attribute's *value*,

cdr a boolean, TRUE indicates that the value is *projected*, FALSE that it is not projected, i.e. *set* from a data load or a manual user *set* operation.

The entire table looks like this:

```
primary hash-table:
 Key        Value
;t          ((Model-Attribute-0 . report?) ... (Model-Attribute-n . report?))

;()         (rule-tuple rule-tuple ...) ; universal rules
                                        ; increasing prec order
;string  i.e. timely-attrribute-name or fact-name
            secondary hash-table:
            Key         Value:
;           t           (value-n . p?)
;           ()          (rule-tuple rule-tuple ...) ; increasing prec order
;           year-0      (value . p?)
;           ...
;           year-n      (value . p?)
Model-attribute-name can be either timely-attributes or facts,
where rule-tuples are of the form:
; (rule-func . precedence)
```

9.1 The INTERNAL-DATABASE-STRUCTURE Package

The INTERNAL-DATABASE-STRUCTURE package provides symbols as per the following package definition:

113 ⟨*internal-data-structure-pkg-def* 113⟩≡ (126)
```
;; provide a package to encapsulate the lowest level of database
;; functionality.
(defpackage "INTERNAL-DATA-STRUCTURE"
  ;; this pkg will access std lisp functions, only.
  (:use "COMMON-LISP")
  (:nicknames "IDS")
  (:export "MAKE-DB"
           "LOOKUP"
           "GET-KEYS"
           "MAP-MODEL-ATTS"
           "GET-MODEL-ATTRIBUTES"
           "GET-RULES"
           ))

(in-package "IDS")
```

9.1.1 `make-db` (&optional (level 1))

This function simply returns a lisp hash-table, with the **:test** set to the lisp function `#'equal`. Depending on the value of the argument, a primary or secondary hash-table is initialized and returned. The argument value of "0" indicates a primary hash-table, any other value indicates secondary hash-table.

The only difference between primary and secondary hash-tables is the initial value that is associated with the key t:

primary hash-table the value t is initialized to *nil*,

secondary hash-table the value t is not initialized.

In both cases the value of the key *nil* is initialized to *nil*.

Arguments:

1. a value, if zero, then create a primary hash-table, otherwise create a secondary hash-table.

Return:

- the properly initialized hash-table.

114 ⟨*make-db* 114⟩≡ (126)
```
(defun make-db (&optional (level 1))
  (let ((ht (make-hash-table :test #'equal)))
    (when (zerop level)
      (setf (gethash t ht) ()))
    (setf (gethash () ht) ())
    ht))
```

9.1.2 `lookup` (key-0 key-1 ht &key val p?)

This is the function providing the SQL-like access to data in the hash-table. It performs both query and updating.

This function will fetch or set the database element depending on arguments provided in the call.

The following semantics are applied to obtain or set a value.

In the following listing, the symbol "fetch" is used to indicate the function that is used to fetch the value. We see that there are only 2 fetching functions and 2 setting functions! How easy can it be!

The "Case ID's" in the listing are references to cases handled by various helper functions that `lookup` uses to perform the work.

```
---------------------------------------------------------------------------
Case
ID   Key-0     key-1    fetch                 set
1: case of a model attribute
     t         n/a      (gethash key-0 ht)    (setf 'fetch
                                               (ordered-instert val 'fetch))
2: case of a universal rule
     ()        n/a      idem                  idem
---------------------------------------------------------------------------
3: case of a factual attribute
     string    t        (gethash key-1        (set 'fetch val)
                          (gethash key-0 ht))
4: case of an attribute's rules
     string    ()       idem                  (setf 'fetch
                                               (ordered-instert val 'fetch))
5:  case of a timely attribute
     string    number   idem                  (set 'fech val)
---------------------------------------------------------------------------
```

Arguments:

1. first-level key: *nil* and *t* are special values, otherwise strings expected. *nil* refers to universal rules; *t* refers to *model attributes*; a string refers to an attribute name,

2. second level key: *nil* and *t* are special, otherwise numbers are expected. *nil* refers to attribute specific rules; *t* refers to the attribute's factual value; a number refers to the year, i.e. second level key to select the attribute's timely value,

3. the hash-table containing the values to be fetched or set,

4. `:val` if not *nil* then set the attribute referenced by the keys to the pair **(val . p?)**, using the above described semantics.

5. `:p?` a precedence or TRUE FALSE indicator to be consed after *:val* depending on whether rules or model attributes are being assigned.

Return: In the case of a fetch operation

1. value,

2. *t* if found, *nil* otherwise.

In the case of a set operation:

1. value,

2. *t*.

The processing in this function is a dispatch to the cases as per described above. We can see that the five cases are reduced to only three dispatches: cases 1 and 2 are assembled, as are cases 3 and 5. Case 4 remains singular.

116 ⟨*lookup* 116⟩≡ (126)

```
(defun lookup (key-0 key-1 ht &key val p?)
(cond
  ((or (eq t key-0)   ; case 1
       (null key-0))  ; case 2
   (gethash-case-1-2 key-0 ht val p?))
  ((null key-1)  ; case 4
   (gethash-case-4 key-0 ht val p?))
  (t ; case 3 & 5
   (gethash-case-3-5 key-0 key-1 ht val p?))))
```

9.1.3 `gethash-case-1-2` (key-0 ht val p?)

This is the processing of cases 1 and 2, i.e. either we are dealing with *model attributes* or *rule tuples.* There isn't much difference in the processing

1. First create the *multiple-value-bind* environment and use the lisp function *gethash* to find the value and presence of a value for the primary hash-table key **key-0**,
2. if there is a value argument provided, then we are in a *set operation*, so cons the second value of the pair, i.e. **p?** onto the first to build the value to be set, "s-val",
3. continuing in the "set" case, set the hash-table value by adding the new value pair to the previous list of values delegating the task of properly inserting to `ordered-insert` for the `WUT` package,
4. finally, in both cases return the appropriate multiple values.

117 ⟨*gethash-case-1-2* 117⟩≡ (126)

```
(defun gethash-case-1-2 (key-0 ht val p?)
  (multiple-value-bind
   (res found?) (gethash key-0 ht)
   (if val
       (let ((s-val (cons val p?)))
         (setf (gethash key-0 ht)
               (wut:ordered-insert s-val res))
         (values s-val t))
     (values res found?))))
```

9.1.4 gethash-case-3-5 (key-0 key-1 ht val p?)

This is the processing of cases 3 and 5, i.e. either we are dealing with a factual or timely attribute. This means that we must deal with the secondary hash-tables, and specifically create one if needed.

1. First, delegate to create (if needed) a secondary hash-table associated with **key-0**, i.e. the attribute string,
2. Now, if there is a **val** argument then use the same logic as in the preceding `gethash-case-1-2` to build the value pair, then associate it but without the need for any insertion into lists since the value is a singleton, and finally return the appropriate multiple values.

118 ⟨*gethash-case-3-5* 118⟩≡ (126)

```
(defun gethash-case-3-5 (key-0 key-1 ht val p?)
  (create-secondary-table key-0 ht)
  (if val
      (let ((s-val (cons val p?)))
        (setf (gethash key-1 (gethash key-0 ht)) s-val)
        (values s-val t))
    (gethash key-1 (gethash key-0 ht))))
```

9.1.5 gethash-case-4 (key-0 ht val p?)

This is the last case, the processing of attribute specific rules. This is a kind of combination of the previous cases.

1. First, delegate to create (if needed) a secondary hash-table associated with **key-0**, i.e. the attribute string,
2. if there is a value argument provided, then we are in a *set operation*, so cons the second value of the pair, i.e. **p?** onto the first to build the value to be set, "s-val",
3. continuing in the "set" case, set the hash-table value by adding the new value pair to the previous list of values delegating the task of properly inserting to `ordered-insert` for the `WUT` package,
4. finally, in both cases return the appropriate multiple values.

119 ⟨*gethash-case-4* 119⟩≡ (126)

```
(defun gethash-case-4 (key-0 ht val p?)
  (create-secondary-table key-0 ht)
  (multiple-value-bind
   (rule-lis found?)
   (gethash () (gethash key-0 ht))
   (if val
       (let ((s-val (cons val p?)))
         (setf (gethash () (gethash key-0 ht))
               (wut:ordered-insert s-val rule-lis))
         (values s-val t))
     (values rule-lis found?))))
```

9.1.6 `create-secondary-table` (key-0 ht)

This little helper function checks for a value for the argument **key-0** in the hash-table **ht** and if no value is found, creates a secondary hash-table and sets it to be the value.

Arguments:

1. a string key value for the the hash-table in second argument,
2. a hash-table perhaps containing a value associated with the first argument.

Return:

- *nil* if hash-table was already available, the hash-table otherwise.

120a ⟨*create-secondary-table* 120a⟩≡ (126)

```
(defun create-secondary-table (key-0 ht)
  (multiple-value-bind
   (unused found?) (gethash key-0 ht)
   (declare (ignore unused))
   (when (not found?)
     (setf (gethash key-0 ht) (make-db)))))
```

9.1.7 `get-keys` (ht)

This simple function simply returns all the keys to the hash-table provided as argument.

Arguments:

1. a hash table.

Return:

- the list of keys in the hash-table.

120b ⟨*get-keys* 120b⟩≡ (126)

```
(defun get-keys (ht)
  (let ((res ()))
    (maphash #'(lambda (key unused)
                 (declare (ignore unused))
                 (push key res))
             ht)
    res))
```

9.1.8 `map-model-atts` (func-one-arg ht &optional report?)

Similar to the lisp mapping functions, this maps the "func-one-arg" (function taking one argument) over all the *model attributes* and returns as multiple values the results as well as the attributes. The optional boolean **report?** if TRUE means that only model attributes with the cdr TRUE will be selected.

Arguments:

1. a function that takes exactly one argument, which is a hash-table containing the values for that attribute, be they fact, rules, or timely. If this argument is null, it is NOT applied,
2. the hash table containing the database,
3. a boolean, if TRUE only model-attribute pairs with cdr TRUE will be selected for the application of the **func-one-arg**.

Return:

1. a list which contains the results of the function calls,
2. the list of the model attributes to which the function was applied.

The processing is done in 2 parts, first build the list of selected *model attributes*, then apply the function to that list and collect all the results.

121 ⟨*map-model-atts* 121⟩≡ (126)

```
(defun map-model-atts (func-one-arg ht &optional report?)
  (let ((model-att-lis (mapcan #'(lambda(pair)
                                   (if report?
                                       (when (cdr pair) (list (car pair)))
                                     (list (car pair))))
                                (lookup t 'unused ht))))
    (values (when func-one-arg
              (mapcar #'(lambda(model-att)
                          (funcall func-one-arg (gethash model-att ht)))
                      model-att-lis))
            model-att-lis)))
```

9.1.9 get-rules (att &key ht)

This function returns a list of all rules in precedence order that apply to the attribute. This includes general and specific rules, or may be limited to only general rules depending on attribute's value. In any case, the multiple values returned handle all the possible cases.

Arguments:

1. attribute as string of could be *nil* if only general rules are requested,
2. **:ht** a hash table (keyword argument).

Return:

1. all applicable rules,
2. specific rules for att.

This is straightforward, the only slight subtlety is the skimming off of the precedence values by means of a mapping of the lisp function *#'car* over the pairs returned by `lookup`.

122 ⟨*get-rules* 122⟩≡ (126)

```
(defun get-rules (att &key ht)
  (let ((gen-rule-lis (ids:lookup  () () ht))
        (specif-rule-lis (ids:lookup  att () ht)))
    (values (mapcar
             #'car
             (wut:map-ordered-insert gen-rule-lis specif-rule-lis))
            (mapcar
             #'car
             specif-rule-lis))))
```

9.1.10 `get-model-attributes` (&key ht report?)

This function returns a list of all *model attributes* selected as per boolean **report?**. If **report?** is TRUE, then only the *model attributes* marked for reporting are returned, otherwise all are returned.

Arguments:

:ht a hash table,

:report? this is used to condition return to only "reporting" *model attributes*. If it is TRUE then only the reporting attributes will be returned, otherwise all *model attributes* will be returned

Return:

- The list of model attributes as per arguments.

The processing work is delegated to `map-model-atts`.

123 ⟨*get-model-attributes* 123⟩≡ (126)

```
(defun get-model-attributes (&key ht report?)
  (multiple-value-bind
   (unused model-att-lis)
   (ids:map-model-atts () ht report?)
   (declare (ignore unused))
   model-att-lis))
```

9.2 Test Harness

124 ⟨*internal-data-structure-test-harness* 124⟩≡ (126)

```
(defparameter *c-c-alis* ())
(setf *c-c-alis*
  '(("Create an Internal DB:"
     .
     "(format t \"~S~%\" (setf db (ids:make-db 0)))")
    ("Declare timely-att-0 to be a model attribute not reported:"
     .
     "(format t \"~S~%\"
              (ids:lookup t () db :val \"timely-att-0\"))")
    ("Declare fact-att-0 to be a model attribute reported:"
     .
     "(format t \"~S~%\"
              (ids:lookup t () db :val \"fact-att-1\" :p? t))")
    ("Insert a general rule-0 prec 0:"
     .
     "(format t \"~S~%\"
              (ids:lookup  () () db :val \"gen-rule-0\" :p? 0))")
    ("Insert a general rule-1 prec 100:"
     .
     "(format t \"~S~%\"
              (ids:lookup  () () db  :val \"gen-rule-1\" :p? 100))")
    ("Insert a \"fact-att-0\" value:\"John\" NOT projected:"
     .
     "(format t \"~S~%\"
              (ids:lookup \"fact-att-0\" t db
                          :val  \"John\" :p? nil))")
    ("Insert a rule for \"fact-att-0\" value:\"fa-0-rule-0\" prec: 20:"
     .
     "(format t \"~S~%\"
              (ids:lookup \"fact-att-0\" () db
                           :val  \"fa-0-rule-0\" :p? 20))")
    ("Insert a rule for \"fact-att-0\" value:\"fa-0-rule-1\" prec: 50:"
     .
     "(format t \"~S~%\"
              (ids:lookup \"fact-att-0\" () db
                          :val  \"fa-0-rule-0\" :p? 50))")
    ("Insert a \"fact-att-1\" \"GOOPY\" PROJECTED:"
     .
     "(format t \"~S~%\"
              (ids:lookup \"fact-att-1\" t db
                          :val  \"GOOPY\" :p? t))")

    ("Insert a \"timely-att-0\" year: 2000 value:200 NOT projected:"
     .
     "(format t \"~S~%\"
              (ids:lookup \"timely-att-0\" 2000 db
                          :val 200 :p? nil))")
```

```
("Insert a \"timely-att-0\" year: 2001 value:201  NOT projected:"
 .
 "(format t \"~S~%\"
          (ids:lookup \"timely-att-0\" 2001 db
                      :val 201 :p? nil))")
("Insert a \"timely-att-0\" year: 2002 value:202 PROJECTED:"
 .
 "(format t \"~S~%\"
          (ids:lookup \"timely-att-0\" 2002 db
                      :val 202 :p? t))")
("Insert a \"timely-att-0\" ta-0-rule-0 prec:30"
 .
 "(format t \"~S~%\"
          (ids:lookup \"timely-att-0\" () db
                      :val  \"ta-0-rule-0\" :p? 30))")
("Insert a \"timely-att-0\" ta-0-rule-1 prec:10"
 .
 "(format t \"~S~%\"
          (ids:lookup \"timely-att-0\" () db
                      :val \"ta-0-rule-1\" :p? 10))")
("Examine the internal data structure:"
 .
 "(format t \"~S~%\" db)")
("Lookup all the name-report? pairs for the model attributes:"
 .
 "(format t \"~S~%\" (ids:lookup  t () db))")
("Lookup all the general rules:"
 .
 "(format t \"~S~%\" (ids:lookup  () () db))")

("Lookup \"fact-att-0\" "
 .
 "(format t \"~S~%\" (ids:lookup \"fact-att-0\" t db))")
("Lookup all rules for \"fact-att-0\" "
 .
 "(format t \"~S~%\" (ids:lookup  \"fact-att-0\" () db))")
("Lookup  \"timely-att-0\" for the year 2000: "
 .
 "(format t \"~S~%\" (ids:lookup \"timely-att-0\" 2000 db))")
("Lookup  \"timely-att-0\" for the year 2001: "
 .
 "(format t \"~S~%\" (ids:lookup \"timely-att-0\" 2001 db))")
("Lookup  \"timely-att-0\" for the year 2002: "
 .
 "(format t \"~S~%\" (ids:lookup \"timely-att-0\" 2002 db))")
("Get all rules for \"timely-att-0\" "
 .
 "(format t \"~S~%\" (ids:lookup \"timely-att-0\" () db))")
("Get all the keys:"
 .
```

```
 "(format t \"~S~%\" (ids:get-keys db))")
("Apply the identity function to all model-attributes hash tables:"
 .
 "(format t \"~A~%\"
     (multiple-value-bind
        (res atts)
        (ids:map-model-atts #'identity db)
        (format nil \"~S~% ~S~%\" res atts)))")
("Apply the identity function to all report? model-attributes:"
 .
 "(format t \"~A~%\"
     (multiple-value-bind
        (res atts)
        (ids:map-model-atts #'identity db t)
        (format nil \"~S~% ~S~%\" res atts)))")
))
```

9.3 Physical Layout of the File

The package is ordered as per the following:

126 ⟨*internal-data-structure.lisp* 126⟩≡

```
;;; internal-data-structure.lisp
⟨lisp-header 143b⟩
⟨internal-data-structure-pkg-def 113⟩
⟨make-db 114⟩
⟨lookup 116⟩
⟨gethash-case-1-2 117⟩
⟨create-secondary-table 120a⟩
⟨gethash-case-3-5 118⟩
⟨gethash-case-4 119⟩
⟨get-keys 120b⟩
⟨map-model-atts 121⟩
⟨get-rules 122⟩
⟨get-model-attributes 123⟩
⟨eoc 143c⟩
⟨internal-data-structure-test-harness 124⟩
⟨eof 144⟩
```

9.4 INTERNAL-DATA-STRUCTURE Package History

2005 12 08: GF creation of the file.

2005 12 11: GF tested ok!

2005 12 18: GF updated to ignore setting of null values, suppression of confusing lookup keyword "set",

2005 12 28: GF update to handle "package-defs.lisp" and correct name-space problems.

2006 01 17: GF update to support the use of boolean in the model-attribute pairs.

2006 02 01: GF made the test harness *C-C-ALIS* private to the IDS package.

2006 02 03: GF minor updates after code-review.

2006 02 09: GF transferred `get-rules` and `get-model-attributes` to this file from stock-data-structure which has become obsolete.

2006 04 05: GF modified second return value of `get-rules` to eliminate the pairs, returning only the rules in correct precedence order.

2006 05 18: GF created create-secondary-table to simplify processing and eliminate duplicated lines of code.

Chapter 10

SAL Utilities

This is the description of the `WIG-UTIL` package providing encapsulation of some more or less general utility functions. Some of these are very general in their nature while others are specialized for the SAL application.

10.1 Code elements

It should be noted that this package cannot load unless the package `sal-config` has already been loaded. An error will be generated if this is attempted.

128 ⟨*utilities package* 128⟩≡ (143a)

```
(defpackage "WIG-UTIL"
  (:use "COMMON-LISP")
  (:nicknames "WUT")
  (:export "2STRING"
           "ABS-YEAR"
           "CURRENT-YEAR"
           "LOG-IT"
           "MAP-ORDERED-INSERT"
           "MAPPEND"
           "NUMLIST"
           "ORDERED-INSERT"
           "OUT-STREAM"
           "PATH-GET"
           "TEST"
           "S-ASSOC"
           ))

(in-package "WIG-UTIL")
```

10.1.1 2string (thing)

This function takes a lisp object and returns a sting representation of it. It converts *nil* into the empty string. The `~A` format instruction is used to perform the conversion.

Arguments:

1. a lisp object.

Return:

- The resulting list.

129a ⟨*2string* 129a⟩≡ (143a)

```
(defun 2string(thing)
  (format nil "~A" (or thing "")))
```

10.1.2 abs-year (y current-year)

Converts a relative date value, *nil* or *t*, into an absolute year value. If the relative date is *nil*, then the result is the current year. If the relative date is *t*, then the result is simply *t*, also.

Dates are considered *relative* iff $y \leq 100$.

Arguments:

1. a date, *nil*, or *t*,
2. the current year as a number.

Return:

- An absolute year value.

129b ⟨*abs-year* 129b⟩≡ (143a)

```
(defun abs-year (y current-year)
  (cond
   ((null y) current-year)
   ((eq y t) t)
   ((> y 100) y)
   (t (+ y current-year))))
```

10.1.3 `cfg-pkg-name-eval` (target-string)

This functions enables compilation and load of files that refer to sal-config package before the file sal-config is loaded. It reads the value of the variable corresponding to the target string from the sal-config package such that the reference can be loaded, without the package being present.

Arguments:

1. the sting name of the variable to be read.

Return:

- the value for the evaluation of that variable.

130a ⟨*cfg-pkg-name-eval* 130a⟩≡ (59 158a)

```
(defun cfg-pkg-name-eval (target-string)
  (eval
   (read-from-string
    (concatenate 'string
                 "sal-cfg:"
                 target-string))))
```

10.1.4 `current-year` ()

Uses system time to return the current year as number.

Arguments: *none.*

Return:

- The current year as a number.

130b ⟨*current-year* 130b⟩≡ (143a)

```
(defun current-year()
  (multiple-value-bind
   (sec min hr date mn yr)
   (get-decoded-time)
   (declare (ignore sec min hr date mn))
   yr))
```

10.1.5 Logging: `log-it` (str &optional (on t))

The logging functionality is handled in a not obvious manner. This is due to the fact that at the end of the main development we realized that logging could be useful, but that the overhead should be controlled by an on/off toggle. The implementation of the toggle, without disrupting the embedded logging calls already in place lead to the implementation described below.

The place where logs are written is controlled by the variables defined in the SAL-CONFIG file.

131 ⟨*log-it* 131⟩≡ (143a)

```
(defun log-it (str &optional (on t))
  (when on (progn (do-log "Logging started.")
                  (do-log str)))
  (setf (symbol-function 'wut:log-it)
   (if on #'logging
     #'not-logging )))
```

The normal use of logging is to send a string to the log by means of a call:

```
(log-it "String to log, with no ending new-line.")
```

Clever readers may have noticed the optional argument `on` which defaults to TRUE. Indeed, at any time the logging can be toggled on or off by means of this second argument.

So how does it work? If we look at the code, we see that the function comprises two parts. First, if the condition `on` is TRUE, then there is a call to the function `do-log`, sending it the message that logging has started. Next, there is another call to `do-log` with the argument `str`, from the invocation of log-it. All this is only if the argument `on` was TRUE.

Now look carefully at the next part which is executed in all cases. The function value of the symbol `log-it` is set to a value depending on the value of `on`. If TRUE, then the function value of `log-it` is set to the function `logging`. Otherwise, it is set to the function `not-logging`.

NOTE: The return value of logging calls should not be considered as significant.

Now you're wondering, well then what happens to the function `log-it` that we're looking at? The answer is simple: **GC**. Yep, it gets garbage collected on the next GC. But, but, but ... how can this ever work? Well, it works because of the code of the next two functions `logging` and `not-logging`.

Both of these take arguments as per `log-it` and as you saw above, they are both *called* as `log-it`, but they are called under different circumstances.

The function `logging` is only called when logging is active. It will always log the value of the argument `str`. Then, if the second *toggle* argument is TRUE, the function simply returns TRUE. However, if the toggle is FALSE, then we again see a re-assignment of the function value of the symbol `log-it`. This time it is assigned to the function `not-logging`.

132 ⟨*logging* 132⟩≡ (143a)

```
(defun logging (str &optional (on t))
  (do-log str)
  (if on t
    (progn (do-log "Logging stopped.")
           (setf (symbol-function 'log-it) #'not-logging))))
```

If `not-logging` is called, under the name of `log-it` then we will see behavior which is just the opposite of that of `logging`.

When it is called, it checks to see if the toggle argument is TRUE. This means that logging must be switched on, the function value of `log-it` re-assigned, and that the message must be logged. This should be clear.

133a ⟨*not-logging* 133a⟩≡ (143a)

```
(defun not-logging (str &optional (on nil))
  (if on (progn
           (setf (symbol-function 'log-it) #'logging)
           (do-log "Logging started.")
           (do-log str))
    t))
```

Let's now look at the actual logging execution. This is handled by the function `do-log`. Remember, that this function is never called by anyone other than by indirection under calls to the name `log-it`.

This function uses the SAL-CONFIG parameters to determine the target for the log output. These can be either std-out, a file, or both.

Each log entry is prepended with the date and time. Log entries should be provided to `do-log` without a trailing new-line since this character is automatically appended on the log message string.

133b ⟨*do-log* 133b⟩≡ (143a)

```
(defun do-log(str)
  (multiple-value-bind
   (sec min hr date mn yr) (get-decoded-time)
   (let ((output
          (format nil
                  "[~A-~2,1,0,'0@A-~2,1,0,'0@A ~2,1,0,'0@A:~2,1,0,'0@A:~2,1,0,'0@A]:  ~A~%"
                   yr mn date hr min sec str)))
     (when sal-cfg:*io-log-to-stdout*
       (format t output))
     (when sal-cfg:*io-log-to-file*
       (with-open-file
        (stream
         (path-get sal-cfg:*io-log-filename-string*
                   sal-cfg:*io-output-path-string*)
         :direction :output
         :if-exists :append
         :if-does-not-exist :create)
        (format stream output))))))
```

10.1.6 `mappend` (fn &rest lists)

This is a non-destructive mapcan.

Arguments (as per mapcan):

1. a function,
2. as many lists as there are arguments to the function.

Return:

- The result of applying the function to the successive cars of the lists, all appended together.

134a ⟨*mappend* 134a⟩≡ (143a)

```
(defun mappend (fn &rest lists)
  (apply #'append (apply #'mapcar fn lists)))
```

10.1.7 `numlist` (start stop &optional (res ()))

This function returns a list of numbers on $[start, stop]$. There is no checking of arguments so be sure that $start \leq stop$!

Arguments:

1. starting point,
2. ending point,
3. accumulative result, not used by caller, only for tail recursion.

Return:

- The resulting list.

134b ⟨*numlist* 134b⟩≡ (143a)

```
(defun numlist (start stop &optional (res ()))
  (declare (integer start stop))
  (cond
   ((> start stop)())
   ((= start stop) (cons stop res))
   (t (numlist start (1- stop) (cons stop res)))))
```

10.1.8 Ordered Insertion of pairs in a-lists

`ordered-insert` (pair tail &optional (reversed-head ()))

Insert a value-pair into an ordered list of those same type of value-pairs. The supported types for val are given below. The updated ordered list is returned. The logic for insertion is:

pairs of type `(function . number)` the list is ordered by increasing numerical order of the cdr's. In case of equality, the value is just inserted, duplicate cdr values are ok.

pairs of type `(string . bool)` all pairs are inserted in the ordering of first-in, first in list.

It is assumed that the `tail` is already ordered according to the above logic. If this is not the case, `map-ordered-insert` below may be used to order the list by means of the call:

```
(map-ordered-insert tail ())
```

All elements must be compatible, there is no error checking.
Arguments:

1. A value-pair of type `(string . genuine-bool)` i.e. either: `(string . nil)` or `(string . t)`
   or of type `(function . number)`.
2. A list of the same type of pair as that given in previous argument,
3. This is a tail-recursion argument used to accumulate the list elements before returning the updated list.

Return:

- The updated list with val-pair properly inserted.

135 ⟨*ordered-insert* 135⟩≡ (143a)

```
(defun ordered-insert (pair tail &optional (reversed-head ()))
  (if (stringp (car pair)) (string-pair-insert pair tail reversed-head)
    (number-pair-insert pair tail reversed-head)))
```

`number-pair-insert` **(pair tail reversed-head)**

Inserts a pair `(something . number)` into an ordered list of those same type of pairs. The logic is that the list is ordered by increasing numerical order of the cdr's. In case of equality, the pair is just inserted, duplicate cdr values are ok.
Arguments:

1. A value-pair of `(function . number)`,
2. A list of the same type of pair as that given in previous argument,
3. This is a tail-recursion argument used to accumulate the list elements before returning the updated list.

Return:

- The updated list with the pair properly inserted.

136a ⟨*number-pair-insert* 136a⟩≡ (143a)

```
(defun number-pair-insert (pair tail reversed-head)
  (if (or (null tail)
          (<= (cdr pair) (cdar tail)))
      (do-insert-return pair tail reversed-head)
    (number-pair-insert pair
                        (cdr tail)
                        (cons (car tail) reversed-head))))
```

`string-pair-insert` **(pair tail unused)**

Inserts a pair `(string . bool)` into a list of the same. Elements are inserted in first-in, first in-list order. Duplicates are filtered. This only works if *all model attributes are inserted before ANY reporting model attributes*!
Arguments:

1. Pair of `(string . bool)` to insert
2. Tail of list into which we insert.
3. Unused.

Return:

- The new list with pair properly inserted.

136b ⟨*string-pair-insert* 136b⟩≡ (143a)

```
(defun string-pair-insert (pair tail unused)
  (declare (ignore unused))
  (let ((tester #'(lambda(l r)
                 (string= (car l) (car r)))))
    (append (remove  pair
                     (remove (list (car pair)) tail :test tester)
                     :test tester)
            (list pair))))
```

`do-insert-return` **(new tail reversed-head)**

Inserts the NEW element at the head of the tail and put the head back in front, returns the resulting list.

Arguments:

1. a lisp object,
2. a list,
3. the reversed head of the list.

Return:

- The list resulting by appending *(reverse reversed-head)* to *(new tail)*.

137a ⟨*do-insert-return* 137a⟩≡ (143a)

```
(defun do-insert-return (new tail reversed-head)
  (append (reverse reversed-head) (cons new tail)))
```

`map-ordered-insert` **(lis-0 lis-1 &optional (res ()))**

Perform the ordered insertion of all the elements in `lis-0` into `lis-1` and return the resulting list. All elements must be compatible, there is no error checking.

Arguments:

1. a list of pairs to be inserted,
2. a list of pairs into which they will be inserted,
3. This is a tail-recursion argument used to accumulate the list elements before returning the updated list.

Return:

- The updated list with all the elements of lis-0 properly inserted.

137b ⟨*map-ordered-insert* 137b⟩≡ (143a)

```
(defun map-ordered-insert (lis-0 lis-1 &optional (res ()))
  (cond
   ((and (null lis-0) (null lis-1) res))
   ((null lis-0) (map-ordered-insert lis-1 () res))
   (t  (map-ordered-insert (cdr lis-0)
                           lis-1
                           (wut:ordered-insert (car lis-0)
                                               res)))))
```

10.1.9 `out-stream` (&key file-name-string path-string)

Return an open stream for writing, superseding any previous one, don't forget to close it when done writing. If either arg is *nil*, then *t* is returned.

Arguments:

1. file name as string
2. path as string

Return:

- either an open stream, or *t*.

138a ⟨*out-stream* 138a⟩≡ (143a)

```
(defun out-stream (&key
                   file-name-string
                   path-string)
  (if (not (and file-name-string path-string)) t
    (open (path-get file-name-string
                    path-string)
          :direction :output
          :if-exists :supersede
          :if-does-not-exist :create )))
```

10.1.10 `path-get` (file-name path-string)

Return a PATHNAME object built from the arguments. There is no error checking.

Arguments:

1. a file-name as string,
2. a path as string ending in slash (or anti-slash on Windoz).

Return:

- a PATHNAME object corresponding to arguments.

138b ⟨*path-get* 138b⟩≡ (143a)

```
(defun path-get (file-name path-string)
  (pathname
   (concatenate 'string
                path-string
                file-name)))
```

10.1.11 Automatic Testing Suite: `test` (pkg-name)

The following variable and functions implement the automatic testing suite.

To test a package, define a package private local parameter named `*c-c-alis*` which contains pairs of the form: `(comment . executable-expression)`.

These pairs should be such that the *comment* describes the action that will take place in the execution of the *cdr*. The test execution function `test`, sequentially displays each *car* on std-out, then it evaluates the *cdr* which should be wrapped in a format statement if the result is to be displayed.

For example, to test the package "TOTO" we would use the following call (note that case of the package name is not significant.):

```
(wut:test "toto")
```

`test` (package-name-as-string)

The function `test` is the entry point to the testing suite. It takes a package name as argument and runs the package's tests. It records the results of execution into a file named "dribble.lisp" and times the entire test for benchmarking purposes.

It uses the function `p-name-2-test-alis` to transform the package name argument into a name of the form `PACKAGE::*c-c-alis*`.

Arguments:

1. a package names as a string.

Return:

- The result of the call to `time` which wraps the mapc over the execution of the pairs in the `*c-c-alis*`.

139 ⟨*test* 139⟩≡ (143a)

```
(defun test (package-name-as-string)
  (let ((cc-alis `(("Starting dribble." . "(wut::drib)")
                   ,@(p-name-2-test-alis package-name-as-string)
                   ("Stopping dribble." . "(wut::drib nil)")
                   )))
    (time
     (progn
       (mapc #'(lambda (pair)
                 (comment-exec pair))
             cc-alis)
       (log-it (format nil "End of test."))))))
```

`comment-exec` **(pair)**

This does the work of printing to std-out the car and cdr of the pair, then executes the cdr. The cdr should have its own format statement if the output is to be visible and dribbled.

Arguments:

1. a pair of strings, the cdr must be an executable lisp statement as a string.

Return:

- The result of evaluating the cdr of the pair.

140a ⟨*comment-exec* 140a⟩≡ (143a)

```
(defun comment-exec (pair)
  (format t "~%;;; ~A~%;;; ~A~%" (car pair) (cdr pair))
  (eval (read-from-string (cdr pair))))
```

`p-name-2-test-alis` **(pname)**

This takes a package name as a string, not case-sensitive and returns the value of the package's *C-C-ALIS*. Yes, I meant the value!

Arguments:

1. a string package name, not case sensitive.

Return:

- the evaluation of PNAME::*C-C-ALIS*.

140b ⟨*p-name-2-test-alis* 140b⟩≡ (143a)

```
(defun p-name-2-test-alis (pname)
  (let ((pn (string-upcase pname)))
    (eval (read-from-string
           (concatenate 'string pn "::*C-C-ALIS*")))))
```

`drib` **(&optional (on? t)(path sal-config:*io-output-path-string*))**

The implementation of dribbling is slightly more subtle than one would expect. This is due to the way lisp considers it an error to turn dribbling on if it is already on, or off it is already off. Despite that little difficulty, this function is straightforward;

The package private parameter *dribbling?* is used to maintain the state of dribbling.

140c ⟨*drib* 140c⟩≡ (143a) 141 ▷

```
(defvar *dribbling?* nil)
```

The function `drib` takes some arguments that can be used to toggle on/off and to direct the output to a specific directory. Note that the default directory depends on the availability of the package `sal-config`.

If dribbling is being turned on, this builds a *pathname* by a call to `path-get`, then logs a "start dribbling" message.

If dribbling is being turned off, then it is stopped and a "Stopped dribbling to wherever" message is logged, and it returns.

Arguments:

1. a boolean toggle TRUE is on, FALSE is off,
2. a string path to where the file `dribble.lisp` should be written.

Return:

- Not Significant.

141 ⟨*drib* 140c⟩+≡ (143a) ◁140c

```
(defun drib (&optional  (on? t) (path sal-config:*io-output-path-string*))
  (let ((pathname (path-get "dribble.lisp" path)))
    (cond
     (on?
      (ignore-errors (dribble))
      (setf *dribbling?* t)
      (dribble pathname)
      (log-it (format nil "Dribbling to ~A" pathname)))
     (*dribbling?*
      (setf *dribbling?* ())
      (dribble)
      (log-it (format nil "Stopped dribbling to ~A" pathname)))
     (t ()))))
```

10.1.12 s-assoc (str-target s-alis)

Super string assoc: works like assoc, but `str-target` must be a string and `s-alis` must contain *pairs* of form *(string . value)* or *(string-list . value)* or *(t . value)*.

`str-target` matches *pair* if either:

- `str-target` and `(car pair)` are same string,
- `str-target` is member of `(car pair)`

Arguments:

1. a string
2. an a-list as per above

Return:

- if match, as per above, the pair in arg-2 that corresponds to match.
- if no match, *nil*.

142 ⟨*s-assoc* 142⟩≡ (143a)

```
(defun s-assoc (str-target s-alis)
  (assoc-if
   #'(lambda(key)
       (or (and (atom key)
                (equal str-target key))
           (and (listp key)
                (member str-target key :test #'equal))))
   s-alis))
```

10.2 Physical Layout of the File

The package is ordered as per the following

143a ⟨*utilities.lisp* 143a⟩≡
```
;;; utilities.lisp
⟨lisp-header 143b⟩
⟨utilities package 128⟩
⟨s-assoc 142⟩
⟨path-get 138b⟩
⟨out-stream 138a⟩
⟨do-log 133b⟩
⟨logging 132⟩
⟨not-logging 133a⟩
⟨log-it 131⟩
⟨numlist 134b⟩
⟨mappend 134a⟩
⟨2string 129a⟩
⟨drib 140c⟩
⟨test 139⟩
⟨comment-exec 140a⟩
⟨p-name-2-test-alis 140b⟩
⟨do-insert-return 137a⟩
⟨number-pair-insert 136a⟩
⟨string-pair-insert 136b⟩
⟨ordered-insert 135⟩
⟨map-ordered-insert 137b⟩
⟨abs-year 129b⟩
⟨current-year 130b⟩
⟨eoc 143c⟩
⟨utilities-test-harness 145⟩
⟨eof 144⟩
```

143b ⟨*lisp-header* 143b⟩≡ (59 98b 110a 126 143a 152e 158a)
```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This file was generated by noweb. Do not edit. Only edit the
;;; source file and regenerate.
;;; author      : Gratefulfrog: gf_at_gratefulfrog_dot_net
;;; license     : GPL http://www.gnu.org/licenses/gpl.html
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

143c ⟨*eoc* 143c⟩≡ (59 98b 126 143a)
```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; End of Code
;;; Test harness follows
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

144 ⟨*eof* 144⟩≡ (59 98b 110a 126 143a 152e 158a)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; End of File
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

10.3 Package Test Harness

145 ⟨*utilities-test-harness* 145⟩≡ (143a)

```
(defparameter *c-c-alis* ())
(setf *c-c-alis*
      '(("Insert into empty."
          .
          "(format t \"~S~%\" (wut::ordered-insert (cons 1 1) ()))")
         ("Insert first:"
          .
          "(format t \"~S~%\" (wut::ordered-insert (cons 1 0)
                                                   (list (cons 1 1) (cons 1 2))))")
         ("Insert middle:"
          .
          "(format t \"~S~%\" (wut::ordered-insert (cons 1 1)
                                                   (list (cons 1 0) (cons 1 2))))")
         ("Insert End:"
          .
          "(format t \"~S~%\" (wut::ordered-insert (cons 1 1)
                                                   (list (cons 1 0) (cons 1 0.5))))")
         ("Insert String:"
          .
          "(format t \"~S~%\" (wut::ordered-insert (list \"abc\")
                              '((\"a\" . t) (\"ab\") (\"abd\" . t) (\"xx\"))))")
         ("Insert pair:"
          .
          "(format t \"~S~%\" (wut::ordered-insert '(\"toto\" . t)
                             '((\"a\" . t)(\"b\") (\"toto\") (\"zeta\" . t))))")
         ("Insert pair:"
          .
          "(format t \"~S~%\" (wut::ordered-insert '(\"toto\")
                             '((\"a\" . t)(\"b\") (\"toto\") (\"zeta\" . t))))")
         ("Insert pair:"
          .
          "(format t \"~S~%\" (wut::ordered-insert '(\"toto\")
                             '((\"a\" . t)(\"b\") (\"toto\" . t) (\"zeta\" . t))))")
         ("Insert pair:"
          .
          "(format t \"~S~%\" (wut::ordered-insert '(\"totot\")
                             '((\"a\" . t)(\"b\") (\"toto\") (\"zeta\" . t))))")
         ("Insert pair:"
          .
          "(format t \"~S~%\" (wut::ordered-insert '(\"toto\")
                             '((\"a\" . t)(\"b\") (\"zeta\" . t))))")
         ("Map Insert :"
          .
          "(format t \"~S~%\"
               (wut:map-ordered-insert (list (cons 1 1) (cons 2 2))
                                       (list (cons 10 1) (cons 20 2))))")
         ))
```

10.4 Utilities Package History

2005 11 18: GF Creation

2005 12 02: GF added funcs to make name conversions for Blakey's sugar factory.

2005 12 11: GF tested ok!

2005 12 16: GF updated ordered-insert to handle strings + pairs; updated tests.

2005 12 18: GF changed string-2-function-name to `sugar-function-name`, it can now handle t and *nil* as arguments.

2005 12 18: GF updated `abs-year` to handle 't' as argument; updated ¡-cdr to handle null cdrs

2005 12 28: GF update to handle package-defs.lisp and correct name-space problems.

2006 01 18: GF update to `<-cdr` to ensure that only numbers compare with numerical `'<'`.

2006 01 19: GF update to `ordered-insert` to handle comparison of t to *nil*, and non duplication of pairs with t or *nil*; it works!

2006 01 24: GF update to create and perfect `log-it`.

2006 01 26: GF update to create remove-package, `current-year`, `out-stream`.

2006 01 27: GF update to create `path-get`.

2006 01 29: GF update to fix general-compare, include `drib`, reposition `path-get` at top of file.

2006 02 01: GF Re-wrote `ordered-insert`, properly, but now only handles pairs! Updated `comment-exec` for extra new-line after comments, put in some explanation of the function. Updated `p-name-2-test-alis` to work with non-exported alists and put in some comments to explain the function. Updated `sugar-function-name` to reject strings with embedded hyphens, e.g. "This-will-be-rejected." Updated remove-double-spaces to handle empty strings and added some explanation.

2006 02 03: GF new version of `log-it` allowing for toggling.

2006 02 07: GF new version of `ordered-insert` to handle report-attributes ordering according to order in defreport.

2006 02 08: GF creation of `s-assoc` for use in a-list parsing.

2006 02 13: GF creation of the literate version of this file for better documentation and maintenance.

2006 02 19: GF first literate version of this file completed and tested OK!

2006 02 22: GF simplified `do-log` to get rid of useless call to concatenate; Corrected a typo; changed order of function in `ordered-insert` section.

Chapter 11

SAL Configuration

This is the description of the SAL-CONFIG package providing encapsulation of all the configuration parameters used in SAL.

This file contains all sal configuration data. This data is divided according to the following types:

- Build & Install data: these are prefixed bi,
- Stock Model data: these are prefixed sm,
- Input-output data: these are prefixed io.

11.1 The SAL-CONFIG Package

The following code chunk defines the package and the exported symbols. All symbols are simple variables which are used in various places throughout the SAL code.

149 ⟨*sal-config package* 149⟩≡ (152e)

```
(defpackage "SAL-CONFIG"
  (:use "COMMON-LISP")
  (:nicknames "SAL-CFG")
  (:export "*SM-MODEL-PATH-STRING*"
           "*SM-MODEL-FILENAME-STRING*"
           "*SM-INDUSTRY-RULEFILE-STRING-ALIST*"
           "*SM-DEFAULT-INDUSTRY*"
           "*IO-OUTPUT-PATH-STRING*"
           "*IO-LOG-FILENAME-STRING*"
           "*IO-LOG-TO-FILE*"
           "*IO-LOG-TO-STDOUT*"
           "*BI-SYS-PATH-STRING*"
           "*BI-SRC-PATH-STRING*"
           "*BI-BIN-EXTENSION-STRING*"
           "*BI-SRC-FILENAME-STRING-LIST*"
```

```
          ))

(in-package "SAL-CONFIG")
```

11.1.1 Build & Install data

Build & Install data are prefixed `bi`.

This is the directory where the all the lisp source files are available:

150a ⟨*build-install data* 150a⟩≡ (152e) 150b ▷

```
(defparameter *bi-src-path-string* "⟨src-file-path 3b⟩")
;  "/home/bob/Desktop/Programming/lisp/StockEvaluator/Work/V7.0/")
```

This is the directory where the all compiled system files will be stored:

150b ⟨*build-install data* 150a⟩+≡ (152e) ◁ 150a 150c ▷

```
(defparameter *bi-sys-path-string* "⟨bin-file-path 3c⟩")
;  "/home/bob/Desktop/Programming/lisp/StockEvaluator/Work/bin/")
```

This parameter contains the extension that the Lisp system uses to identify binary lisp files. This definition technique will work for CMU and GNU Common Lisp. For other Lisps, please update appropriately.

150c ⟨*build-install data* 150a⟩+≡ (152e) ◁ 150b 150d ▷

```
(defparameter *bi-bin-extension-string*
  (let ((sys (lisp-implementation-type)))
    (cond
     ((string-equal sys "CLISP") ".fas")
     ((string-equal sys "CMU Common Lisp" ) ".x86f")
     ( (error "Unknown Lisp implementation type! ~S~%" sys)))))
```

This is the list of all the source files in system. There should be no need to update this parameter.

150d ⟨*build-install data* 150a⟩+≡ (152e) ◁ 150c

```
(defparameter *bi-src-filename-string-list*
  '("utilities.lisp"
    "internal-data-structure.lisp"
    "sugar.lisp"
    "rule-funcs.lisp"
    "sal.lisp"))
```

11.1.2 Stock Model Data

Stock Model data are prefixed `sm`. They are configurable and will require user settings. The actual values given below are examples. Some will certainly have to be changed for the installed system, others may be ok as they stand.

The first defines the path to the *directory* where all model and rule files are stored.

151a ⟨*stock-model-data* 151a⟩≡ (152e) 151b ▷

```
(defparameter *sm-model-path-string* "⟨model-rule-path 3d⟩")
;  "/home/bob/Desktop/Programming/lisp/StockEvaluator/Work/Examples/")
```

The next parameter is the name of the file containing the definition of the model attributes.

151b ⟨*stock-model-data* 151a⟩+≡ (152e) ◁ 151a 151c ▷

```
(defparameter *sm-model-filename-string* "⟨mode-filename 4a⟩")
;  "model.lisp")
```

This next variable is an a-list mapping industries to rule-files. The keys and the file-names *ARE* case sensitive. `CAR` values may be *t*, a single string, or a list of strings. `CDR` values may be both single file-names and lists of file-names. A key value of *t* indicates that the rule-file(s) in the cdr should be loaded in all cases.

151c ⟨*stock-model-data* 151a⟩+≡ (152e) ◁ 151b 151d ▷

```
(defparameter *sm-industry-rulefile-string-alist*
  '(("BANK" . "bank-rules.lisp")
    (("COMPUTER" "MANUFACTURING") . ("computer-rules.lisp"
                                     "manufacturing-rules.lisp"))
    (t . "default-rules.lisp")))
```

This next parameter defines the the default value for the industry, in case the data loaded doesn't define an industry. This will *NOT* stop a unknown industry from being defined. If that happens, only the default rules as defined by the key *t* in `*sm-industry-rulefile-string-alist*` will be loaded.

151d ⟨*stock-model-data* 151a⟩+≡ (152e) ◁ 151c

```
(defparameter *sm-default-industry* "COMPUTER")
```

11.1.3 Input & Output data

Input and output data are prefixed `io`. These are configurable and will require user settings.

This is the path to the directory where the system will write any output during execution:

152a ⟨*input-output data* 152a⟩≡ (152e) 152b ▷
```
(defparameter *io-output-path-string* "⟨io-path 4b⟩")
;  "/home/bob/Desktop/Programming/lisp/StockEvaluator/Work/Output/")
```

This is the file where the system will write any log output during execution. This file will be written to the directory specified in `*io-output-path-string*`.

152b ⟨*input-output data* 152a⟩+≡ (152e) ◁ 152a 152c ▷
```
(defparameter *io-log-filename-string* "⟨log-filename 4c⟩")
;  "sal-log.out")
```

If the following parameter is set to *t*, then all logs will be written to the `*io-log-filename*`. Note: this parameter and `*io-log-to-stdout*` are not mutually exclusive. Any combination of TRUE and FALSE values is acceptable for them.

152c ⟨*input-output data* 152a⟩+≡ (152e) ◁ 152b 152d ▷
```
(defparameter *io-log-to-file* t)
```

If the following parameter is set to *t*, then all logs will be written to std-out.

152d ⟨*input-output data* 152a⟩+≡ (152e) ◁ 152c
```
(defparameter *io-log-to-stdout*  nil)
```

11.2 Physical Layout of the File

The package is ordered as per the following

152e ⟨*sal-config.lisp* 152e⟩≡
```
;;; sal-config.lisp
⟨lisp-header 143b⟩
⟨sal-config package 149⟩
⟨stock-model-data 151a⟩
⟨input-output data 152a⟩
⟨build-install data 150a⟩
⟨eof 144⟩
```

11.3 Sal-Config Package History

2006 02 03: GF minor updates after code-review.

2006 02 08: GF added industry- rule-file associations to test multiple associations.

2006 02 09: GF updated documentation of `*sm-industry-rulefile-string-alist*`. Removed reference to the file stock-data-structure.lisp which has become obsolete.

Chapter 12

The Sal Builder

The SAL-BUILD packages provides building and installation support for SAL.

12.1 The SAL-BUILD Package

The package exports only one symbol: `make-install`. The end user will probably never call this function directly. It is called in the SAL Makefile (cf. section 5 on page 32).

154 ⟨*sal-build-pkg-def* 154⟩≡ (158a)

```
(defpackage "SAL-BUILD"
  ;; provide a package to encapsulate the database functionality
  (:nicknames "SAL-BLD")
  (:use "COMMON-LISP")
  (:export "MAKE-INSTALL"
           ))

(in-package "SAL-BUILD")
```

12.1.1 `load-config` (cfg-pathname)

This function is defined below. The reader should note that this function is also defined in the SAL package (cf. section 6.6 on page 59)[1].

Argument:

1. Full pathname, NOT STRING path, to config file.

Return:

- *t* if successful, *nil* if failure

This will attempt to load the config file. If it is not found, the user is prompted to enter a new path to the config file.

155 ⟨*load-config* 155⟩≡ (59 158a)

```
(defun load-config (cfg-pathname)
  (if (probe-file cfg-pathname)
      (load cfg-pathname)
    (progn (format t "Config file not found:  ~A~%" cfg-pathname)
           (format t "Enter full path to config file or return to exit:~%")
           (format t "> ")
           (let ((new-path (read-line)))
             (if (string= new-path "") nil
               (load-config (pathname new-path)))))))
```

[1] Literate Programming tools allow us to define the function only once and then to include it in as many source files as required!

12.1.2 `make-install` (config-file-pathname-string &key (verbose t))

This function first loads the config file. It then compiles all the SAL source files as defined in the config file. Finally, it installs the binaries in the target directories and loads them.

Arguments: Arguments:

1. full path string to the sal-config file.
2. `:verbose` optional, if *t*, then compiling and loading output are sent to std-out.

Return:

- the extension of the compiled file,
- the list of files installed.

The compiling and loading are delegated to `do-compile-load`.

156 ⟨*make-install* 156⟩≡ (158a)

```
(defun make-install  (config-file-pathname-string &key (verbose t))
  (load-config (pathname config-file-pathname-string))
  (values
   (do-compile-load
    :src-file-string (car (cfg-pkg-name-eval "*bi-src-filename-string-list*"))
    :src-path-string (cfg-pkg-name-eval "*bi-src-path-string*")
    :dest-path-string (cfg-pkg-name-eval "*bi-sys-path-string*")
    :verbose verbose)
   (mapc #'(lambda(f)
             (do-compile-load
              :src-file-string f
              :src-path-string (cfg-pkg-name-eval "*bi-src-path-string*")
              :dest-path-string (cfg-pkg-name-eval "*bi-sys-path-string*")
              :verbose verbose))
         (cfg-pkg-name-eval "*bi-src-filename-string-list*"))))
```

12.1.3 `do-compile-load` (<**args**>)

Keyword Arguments:

:src-file-string a string name of the file to compile,

:src-path-string a string path to the file to compile,

:dest-path-string a string path to the target directory for the compiled file to reside.

:verbose bool if *t* verbosely compile and load, otherwise silent.

This function compiles the file argument from the src path, into the dest path, with verbose as per argument. It then loads the compiled file. It returns the extension of the compiled file.

Return:

- extension of the compiled file, including the leading "." e.g. **".fas"**.

157 ⟨*do-compile-load* 157⟩≡ (158a)

```
(defun do-compile-load (&key
                        src-file-string
                        src-path-string
                        dest-path-string
                        verbose)
  (let ((pathname (compile-file
                   (pathname
                    (concatenate 'string
                                 src-path-string
                                 src-file-string))
                   :output-file (pathname dest-path-string)
                   :verbose verbose
                   :print verbose)))
    (load pathname :verbose verbose :print verbose)
    (pathname-type pathname)))
```

12.2 Physical Layout of the File

The package is ordered as per the following:

158a ⟨*sal-build.lisp* 158a⟩≡

```
;;; sal-build.lisp
⟨lisp-header 143b⟩
⟨sal-build-pkg-def 154⟩
⟨sal-build-debugging-helpers 158b⟩
⟨load-config 155⟩
⟨do-compile-load 157⟩
⟨cfg-pkg-name-eval 130a⟩
⟨make-install 156⟩
⟨eof 144⟩
```

158b ⟨*sal-build-debugging-helpers* 158b⟩≡ (158a)

```
;(load "/home/bob/Desktop/Programming/lisp/StockEvaluator/Work/V7.0/sal-build.lisp")
```

12.3 SAL-BUILD Package History

2006 01 23: GF creation of the file.

2006 01 24: GF creation of load-config, make-install, sal.

2006 01 26: GF update do-compile to do-compile-load

2006 01 29: GF looks good! can build and load without problem.

2006 02 03: GF minor updates after code-review.

2006 02 06: GF minor updates, added ":print verbose" in calls to "load."

2006 03 11: GF update to make config parameters arguments to make-install, and to eliminate the need for pre-loading of sal-config file.

Chapter 13

Outstanding Issues

This list traces the defects and other issues that were encountered during the development of SAL.

1. CANCELLED:
   Use the automatic documentation system create-user-manual to document the source, but read the instructions first!
2. DONE:
   Write "How-to configure files prior to building, or when moving system."
3. DONE:
   Write "How-to `make-install`."
4. DONE:
   Write "How-To Write a simple Rule".
5. DONE:
   Write "How-To Write a generic (multi-attribute) Rule".
6. DONE:
   Re-write the quick start section after most of these changes have been proven correct.
7. CANCELLED:
   update installation instructions to include the setting of the config file name and path in:
   (a) `sal.lisp,`
   (b) sal-build.lisp,
   (c) `sal-config.lisp.`
8. DONE:
   Create testing targets in makefile and automates these by inclusion of

noweb chunks appropriately. Finish this for (`wut:test ‘‘sal’’`) makefile needs to be finalized, using a loop for all the tests. Test targets could be:

(a) all-test,
(b) sal-test,
(c) ids-test,
(d) etc.

9. DONE:
integrate the config path into the literate program so that it loads directly. This will mean 2-phased make: make makefile, then make ; make install.

10. DONE:
be sure to share the definition of load-config in both build-sal.lisp and sal.lisp, and to share the definition of the config-path in build-sal.lisp, sal.lisp and in the makefile.

11. DONE:
Fix error: “`Unknown control sequences: \nobreakspace`” on make doc.

12. CANCELLED:
Integrate the literate programming technique into this document to improve the link between documentation and source code.

13. DONE:
clean up utilities.lisp, move some things to sugar.lisp, document in literate style.

14. DONE in v6.9.1:
Restructure directories for devt. separating all Examples and Config from development.

15. DONE in V6.9:
Repair defect in create-sugar-function where
`(attribute-sugar-function ‘‘CURRENT YEAR’’)` is called, but in fact not evaluated to return the current year value. This means wrapping the call in a `funcall` call. No big problem, but a real defect!

16. DONE in V6.9:
Create `sgr:industry` placeholder function.

17. DONE in V6.9:
Provide `out-file-name` argument for `sal:process` find solution for file names for tickers of odd form.

18. DONE:
Provide script to configure paths at each installation.

19. DONE:
Improve sal:process and sal:report simplify and separate functionality

20. DONE:
Remove stock-data-structure completely from SAL, move functions to ids, correct all calls in sugar.lisp and sal.lisp. Retest fully!

21. DONE:
Enable multiple industry to multiple rule file association in processing of sal-config.lisp

22. DONE:
SEE 25 below for way of doing this with a closure and a function! Provide loop detection in rule firings: This remains difficult to implement in an elegant manner. I hesitate to use a package global variable. I also hesitate to put a new slot in the stock-database-struct, and change all the calls to the parse functions to use the dbs instead of the ht as argument. It seems hard to find a good solution. Suppose that exec-sugar knows about a package global-variable if the value is zero just before the call to funcall, then the table with rule-firings is reset to empty. Then the counter is incremented at each funcall, and decrements it on return. If the value is Zero then the rule-firings table is reset to empty. Meanwhile, in somerule, we check that the rule being fired with its arguments is not already in the rule-firing table, if it is we abort the execution. This should be integrated into the stock-database-struct, along with the sugar-function-lis.

23. DONE:
load report attributes in list order provided by call to defreport. One potentially possible way to do this would be to use the cdr of the model-attributes to be an ordering value, i.e. 0, 1, 2... instead of simply 'T'. Then on report generation, the reporting attributes could be sorted prior to mapping over them. This looks like it requires some mall modifications

    (a) sgr:parse-2-model to handle the order number,
    (b) rf:defreport to pass the order number,
    (c) sds:get-model-attributes to sort on cdrs, by passing the #'function that returns pair if cdr is number or nil otherwise function to map-model attributes, then filtering the results for not null, to get the pairs of (report-att . order-number), then sorting on cdrs.
    (d) sal:process to call the sorted version of get-model-attributes list on the cdrs?

24. CANCELLED:
This has proven to be infeasible after testing in `sal.1.lisp` and `sal-build.1.lisp`. See if it's not possible to use compiled rule-files. This is a modification to the load-rules-helper to (load (compile-file file.lisp) and logit! all this in `sal.lisp`.

In `sal.lisp`, line 145 we are loading the src version of the rule files, not the compiled version as one may have hoped. Why is this? can it be improved? If it can not be changed, then update comments in `key-2-pathname-lis`.

25. DONE:
used a closure to encapsulate the dbs and thus enable cleanup of the sugar functions without a global variable! This may be the way to handle loop detection too.
Remove the *sugar-function-lis* from sugar.lisp and use keys + model attribute instead as means for tracking sugar functions. This simplifies the tracking and is far more elegant.

26. perform all the corrections resulting from the code-review:

    (a) DONE:
    fixed a lot of things! improved performance too!
    corrected bug in sugar-function-creation,
    sugar.lisp

    (b) DONE:
    computer-rules.lisp

    (c) DONE:
    default-rules.lisp

    (d) DONE:
    internal-data-structure.lisp

    (e) DONE:
    model.lisp

    (f) DONE:
    rule-funcs.lisp

    (g) ALL DONE
    DONE:
    logging toggle switch in sal:process, page 21 of print-out,
    sal.lisp

    (h) DONE:
    sal-build.lisp

    (i) DONE:
    sal-config.lisp

    (j) DONE:
    stock-data-structure.lisp

    (k) DONE:
    utilities.lisp

27. DONE:
create an exported function in sugar package to get the sugar function from the attribute name string

```
(defun attribute-sugar-function-from-name (att-name)
   (symbol-function (sugar-function-symbol att-name)))
```

28. DONE:
    Improve logging, with toggle as per log-it.lisp in V6.6. Also, find a way of logging the rule firing for debugging purposes.

29. DONE:
    Update sal:process to inform if zero data are loaded.

30. DONE:
    Provide fault tolerance during an analysis run, as well as logging of all analysis to a log file. Use "error" everywhere since Blakey catches all errors in his own processing!

31. DONE:
    detect embedded hyphens in attribute names and reject for sugar function creation.

32. DONE:
    repair or re-write ordered-insert so that it can be understood by humans,

33. DONE:
    make all *C-C-ALIS* private package variables.

34. DONE:
    But changed to write a default tuple ("INDUSTRY *default-industry-from-config*)
    This Write default-industry rule so that a rule-file will always be loaded, even if the data tuple `(industry ''toto'')` is absent.

35. DONE:
    Write interface functions to Blake's database calls:

    ```
    (blake-get-data (string-downcase (sgr:ticker t))
    (blake-get-rule-file-name (string-downcase (sgr:industry t))
    ```

36. DONE:
    Replace all information calls to `format` with logging calls! Consider logging output choices like `*standard-output*` `*terminal-io*` `*error-output*`.

37. DONE:
    Cleanup

    (a) DONE:
        "sal-config.lisp"

    (b) DONE:
        "sal-build.lisp"

(c) DONE:
"utilities.lisp"

(d) DONE:
"internal-data-structure.lisp"

(e) DONE:
"stock-data-structure.lisp"

(f) DONE:
"sugar.lisp"

(g) DONE:
"rule-funcs.lisp"

(h) DONE:
"sal.lisp"

38. DONE:
IMPORTANT to make the loading/unloading of the rule-funcs package work! Update the make-install to handle:

    (a) system + rule files ??? is this still needed?

    (b) only the rule files. ??? is this still needed?

39. Cancelled:
Find a way of compiling the rule-funcs into their directory!

40. DONE:
Some strange differences in behavior between the CMU and Gnu lisps has caused some setbacks... Setup installation, i.e. the loading from a single file load, embedding the call to load "`sal-config.lisp`". This needs to be documented in the README, or INSTALL file.

41. CANCELLED:
Set verbose loading and compiling by using dynamic extent (shadowing) of the std variables `*compile-verbose*` and `*load-verbose*` by means of a `let`.

42. DONE:
Restructure all the top levels: blake-api, loader, data, rules, etc.

43. DONE:
The loading mechanism should follow the logic:

    (a) init

    ```
    (load "loader")
    ```

    (b) iteration

```
    etc.
    (analyze <args>)
    (goto b)

(defun analyze (ticker start stop \&rest other)
  (clear)
  (defdata (blake-get-data ticker)) ; should return the list as per ibm.lisp
  (defmodel (blake-get-model (sgr:industry t)))
  (load (compile (industry-2-rule-file-name (sgr:industry t)))
  ;; rule file contains calls to defrule, and defreport
  (report)
```

In `blake-api.lisp`, the `load-datum` function *could* be split up to isolate the loading of:

- attribute tuples,
- model-element tuples,
- report-element tuples,
- rule tuples.

44. DONE:
Fix the bad calls to reduce in report as per 6.2-full.

45. DONE:
Fix insertion of model attributes and report attributes! Something is wrong with the insertion of the model attributes and the report attributes, only the model attributes appear to be in the db!

46. DONE:
Create make-install.

47. DONE:
Use the function `probe` to detect presence of config file in: "$HOME/.sal-config.lisp". Obviously, the path must be specified in the loader-file, but that's the only place.

48. DONE:
Implement a logging mechanism that handles both std-out and file based logging.

49. DONE:
Create configuration file `sal-config.lisp`, with at least the following:

```
(defparameter *sys-path*
(defparameter *bin-extension*
(defparameter *model-filename*
(defparameter *industry-rulefile-alist*
```

```
(defparameter *default-rule-filename*
(defparameter *output-path*
(defparameter *log-filename*
(defparameter *src-path*
(defparameter *src-filename-list*
etc.
```

50. DONE:
THIS IS NOT TO BE CHANGED, The semantics of `(sugar ())` is correct and the return value is correct also: *"The second return value for* `(sugar ())` *is not consistent with the first return value. This is not a problem since the value is not currently used."*

51. Refine the language aspect of the system, loading mechanism and restructure. The language elements could be:

    (a) DONE:
    defdata: works!!

    (b) DONE:
    defrule: works!

    (c) DONE:
    defmodel: works!!!

    (d) DONE:
    defreport: works!

52. DONE:
The use of the "model attributes" as report indicators causes the report ones to be duplicated in the hash table at key t –, there is no impact but it's ugly.

53. DONE:
Update Internal-Data-structure and Stock-db to support the use of the cdr in the model attribute pairs as a boolean indicating if the model attribute should be reported, or not. This boolean value will be assigned in the call to `defreport`.

54. DONE:
Resolve the issue of package scoping in the macro calls to `defrule`, `def...`

55. DONE:
Write the following macro to do all the work needed in defining a rule and associating it with attributes and precedence.

```
(defmacro defrule (name
                   applicable-attribute-lis
                   precedence
```

```
                          ;; let the user call the args what he chooses!
                          arg-lis
                          &rest body)
   ...)
```

56. DONE:
    The sugar functions will not be re-defined on a second load of data for attributes that are not in the second data set. This will lead to subtle bugs since the sugar function will work on the wrong internal data structure.

Chapter 14

Index

14.1 Symbol Definition Index

14.2 Defined Code Chunks