

Assembly Language Extensions of Visual Prolog For Data Mining of Huge Data

George A. Stathis (ENB Ltd)

ABSTRACT

Pure Assembly Language is the fastest way to implement Visual Prolog predicates that can achieve intelligent processing of huge data. High-level logic programming power is then combined with extremely low-level speed enhancements. A number of special techniques emerge that combine the best of both worlds: *Visual Prolog and Assembler*.

INTRODUCTION

One important reason that discourages the use of Prolog (and Visual Prolog) in industry is the inherent difficulty of traditional *recursive* Prolog programming techniques to deal with *huge data* adequately. Most Prolog textbooks are prolific with quite elegant list-processing predicates that are adequate for lists of a few hundred (or a few thousand) elements. When the number of list-elements exceeds a certain limit, most of these ‘elegant’ predicates (even using the best “tail recursion optimization”) tend to cause *crashes*. Mainstream languages (such as Visual Basic or C++/C) tend to have simple ways of avoiding such problems (the use of iteration instead of recursion, etc.) that reinforce certain common prejudices against the use of Prolog, while underestimating the advantages and the high-level expressiveness of Prolog (beyond doubt in academic circles).

In this paper we present an overview of very specific methodologies that were used in real-world problems, producing quite adequate -or even *impressive*- results by the *combined use of Visual Prolog and Assembly Language*.

Combining Visual Prolog with Assembly (or with *any* other programming language) is not at all difficult to implement, from the point of view of *linking*. Any predicates implemented in another programming language and linked to Visual Prolog appear and behave *exactly as if* they were implemented in Visual Prolog.

Visual Prolog has been designed with serious optimizations in mind: Whenever possible, it tries to avoid unnecessary computations (that other Prolog compilers perform all the time) if these computations are not required for the operation of each specific (e.g. deterministic) predicate. All linked “foreign language code” works seamlessly inside native Prolog code. As a result, the linking of Assembly language subroutines “as if they are Prolog predicates” is efficient from the outset, and these foreign predicates run as fast as they would in any other (“mainstream”) developer environment.

PREDICATE LIBRARIES

At the time of writing (March 2006) there are more than 1100 Visual Prolog predicates that have been implemented in Assembler, as well as more than 150 predicates implemented as ‘C’ glue-functions. In addition, about 20 new predicates are added to these libraries every month (depending on the practical needs of programming projects). There are families of such predicates dealing with lists, strings and characters, “binaries”, CSV Excel-files (texts delimited by separators), time-series, dates, tree-like data structures and other objects. All these predicates originated from *real needs* in real software applications. The efficiency and speed of these applications destroys the myth of Prolog being “just an academic language”, while outperforming commercial applications of similar nature developed using mainstream languages by a significant factor, typically of an order of magnitude - and quite often *more*.

WORK ORGANIZATION

This work has been organized on the basis of the fact that Assembly Language code is *not* easy to modify or debug unless it is *properly commented*, and it is not useable at all unless the comments include *instructions* for calling each routine externally *as a Prolog predicate*.

CONVERSIONS OF STRINGS TO LISTS

Such conversions are typically necessary for 95% of current Prolog projects. The growing need for such predicates led to about a dozen of them implemented in Assembly. Their full global *definitions* are listed in Appendix F: This library includes customized predicates implementing many such conversions, from strings to lists (and vice versa) in a variety of possible ways.

When dealing with *large texts*, our most basic need is to copy them inside main memory for further processing. Unfortunately, it has been verified that (after certain specific list or text size limits) almost *any* Prolog predicate that uses recursion is *useless for simple tasks, like converting texts to lists!*

This situation *can* (of course) be improved: If we don't want to use Assembly language, it is certainly possible to create optimized Visual Prolog predicates that minimize the use of the stack, so that texts of *somewhat* larger sizes (than before) can be converted, at *somewhat* improved speeds.

Visual Prolog includes implementations of (non-ISO) predicates for string-handling that can access the memory of strings (or lists) *directly*. Examples of code using these built-in predicates are *already given* inside Visual Prolog's *documentation for advanced users*, so they won't be repeated here. The important thing, however, is that even such optimized coding, impossible to do in *most* other Prolog compilers (which lack those extra predicates) and possible *only in Visual Prolog*, inevitably and very quickly hit against some *new limits* of speed and memory, that can be shown to be (this time) *unsurpassable*.

One of our *most highly optimized* conversion predicates is '**strx2list**': It can convert strings (typically *large texts*, which can be of several megabytes size) to *Visual Prolog string-lists*. Here is a table of *measured execution times*, for the optimized predicate "**strx2list**", in all its possible *variants* and *memory methods*.

Size (Mb):	List-Size elements	(A) stack alloc.	(B) Heap alloc.	(C) Pre-alloc.
1	10,000	0.01	<0.01	<0.01
2	200,000	0.02	0.01	<0.01
3	300,000	0.03	0.01	0.02
4	400,000	0.03	0.03	0.02
5	500,000	Crash!	0.03	0.02
6	600,000	-	0.05	0.03
7	700,000	-	0.06	0.03
8	800,000	-	0.06	0.03
...
10	1,000,000	-	0.09	0.05
20	2,000,000	-	0.17	0.09
30	3,000,000	-	0.26	0.14
40	4,000,000	-	0.36	0.18
50	5,000,000	-	0.47	0.23
60	6,000,000	-	0.55	0.28
70	7,000,000	-	0.66	0.36
80	8,000,000	-	0.75	0.39
90	9,000,000	-	0.84	0.44
100	10,000,000	-	0.88	0.48
110	11,000,000	-	1.00	0.50
120	12,000,000	-	1.02	0.52
130	13,000,000	-	1.17	0.61
140	14,000,000	-	1.29	0.64
150	15,000,000	-	1.36	0.72
160	16,000,000	-	1.45	0.74
170	17,000,000	-	1.56	0.80
180	18,000,000	-	2.31	1.48
190	19,000,000	-	3.30	2.12
200	20,000,000	-	8.06	5.08
...
250	25,000,000	-	20.45	19.58

These **timings** show an apparently *impressive speed*: -It is evidently possible to convert a 100Mb text of 10 million lines to a list (of 10 million elements) in *much less* than 1 second!

The timings (in seconds) of columns A, B, C refer to *literally millions or tens of millions* of elements! Alas, it was discovered that (after a *few hundred thousand elements*) most high-level language code *does not work* (Prolog, as well as *any* language). Optimised Prolog code executes at about *one order of magnitude* (i.e. 10 times) *slower than optimised Assembly* for the restricted small range of text and list sizes where it *does work*.

Now, although these results *are correct* (even at higher data-sizes than those shown in the table) at a turning point of about 250 Mb of text (and 25 million elements) the execution time of the ASM-predicates increases steeply and *unpredictably*, due to RAM limitations of the *specific* computer used (P4, 512 Mb).

Experienced users of Visual Prolog may have noticed that columns **A**, **B**, **C** (of the previous timing-table) carry annotations of the *memory allocation method* used in each case. And the three possible ways of allocating memory are **A**) Visual Prolog's stack, **B**) Visual Prolog's heap, and **C**) no *new* allocation (i.e. memory must be *already allocated*, before the call).

All these three *memory allocation methods* are possible, using the predicate "**strx2list**": It was implemented in *two* variants, one with *two* and one with *three* input-arguments.

```
GLOBAL PREDICATES
SLIST strx2list (STRING, CHAR)
      -(i, i) language c
SLIST strx2list (STRING, CHAR, BINARY)
      -(i, i, i) language c
```

Our main concern is producing an output-list which is correct and *adequate* for fast further processing. The inputs are *recycled*, in a very *destructive "procedural" way* -that can make a *Prolog purist* turn his head away in *dismay*:

The first variant **strx2list/2** *allocates memory* before returning the output-list, but... only as *much memory as strictly necessary*. In order to operate well under "extreme conditions", it does *not* allocate any memory for the strings inside the list: Instead (like the 2nd predicate) it *re-uses* the input string in "recycled form" by chopping it up into pieces, while inserting *pointers* to these pieces inside the list, rather than creating new strings and copying the old.

If you are somewhat familiar with Assembly language, you might see what is going on in detail by looking into Appendix **A**, the *source code* of "**strx2list**": The assembly language main loop checks out every byte of the input string for presence of a "separator-character" (arg. 2) while *also* planting into this position

a *zero-byte* (signifier of the "end of a string" in 'C' and Visual Prolog) and then inserts as a list-element *only a pointer* to the start of each *chopped part* of the string (rather than a new string). Consequently, we could say that the input-text is "*recycled*", to play the part of the output-strings (in the list) *as well*.

This is a raw *memory allocation strategy* that works very well to *economise memory*, when processing *huge* texts and lists in **RAM**.

As regards the internal structure of a "list" in Visual Prolog, suffice it to say that it consists of 12-byte structures, where the first 4 bytes are indicators of "element-type" ("normal" or end elements), the next 4 bytes are (pointers to) each element itself, and the last 4 bytes are pointers to the "*next list-element*". So, instead of allocating small pieces of memory N times (for N list-members), memory can be allocated *only once*, for an entire list, if the number of list-elements is *known*: For this reason, the first predicate **strx2list/2** has *two* internal loops: (1) One for finding the number of lines in the text (which is also the number of list-elements) and (2) a second loop that *uses* this number to *create each list-element*. Before the second loop begins, a *single call* of an external memory allocation predicate is enough to reserve memory for the *entire list* (*either* in the stack, *or* in the heap).

The second variant of **strx2list/3** was created for those rare cases when we need to call this predicate several times, inside another loop (external to it). In this case, it is *not* really worth allocating memory during *every* call. Instead, memory is allocated *prior* to the outer loop *only once*, to create a binary buffer which is used by **strx2list/3** as a third (bound) argument, a Visual Prolog "*binary*", pre-allocated with **12*(N+1)** bytes, where **N** is the (pre-calculated) *number of list-elements*. So, the second variant, **strx2list/3**, uses *no* memory allocation at all. Instead of this, it *recycles all* the memory of its *own inputs*. The reason we use a "binary" is that Visual Prolog binaries are equipped with an *indicator of size* embedded in a memory location *before* their

pointer. This size-indicator can be used later to *release the heap-memory* (if in the heap) precisely as required, without side-effects that can arise from intermediate processing (e.g. zero-bytes inside a string, making it appear shorter, confusingly for *de-allocation*, etc).

Of course, there are other similar conversion predicates in this family that *don't* behave as *stingily* towards memory, as does “**strx2list**”: Predicates “**str2list**” (with no “x” before the 2) and “**conclist/2**” are routinely used in most ordinary *text-to-list / list-to-text conversions*.

In Appendix **B**, a full list of Global Predicate specifications is given for *all* list-conversion predicates in these libraries, as well as many other *list-handling* ASM predicates. Some of these predicates are *non-deterministic*: There are interesting *non-deterministic versions of list-membership* and other list-operations in this library that *cannot* be discussed here, due to limitations of space and time.

As a final note (about the predicate **strx2list**) that may be of general value, if you examine the source code of **strx2list** (in Appendix **A**), you may be surprised to realize that it doesn't call *literally* any external memory *allocation predicates*. Instead of this, it uses *function-pointers* to an external allocation predicate, included in the library's *Global data segment* as “**_allocfunc**”. This was done for *flexibility*; so that the *actual* allocation predicate may be changed, at any time. E.g. the test-program, which produced the results for the previous *timing-table*, used different assignments for “**_allocfunc**” for each column of the results:

Column **A** was generated while using (as the *actual contents* of the location “**_allocfunc**”) Visual Prolog's stack-allocation predicate “**_MEM_AllocGStack**”, and column **B** was created by using another -similar- predicate, which allocates memory in the *heap*, instead. The results (shown in the timing table) should *not* surprise experienced Visual Prolog users:

- Heap memory is *much more* suitable -than the stack- for storing *huge data*!

In conclusion, programming techniques such as these may seem “dirty” to “Prolog Purists” (since they are “procedural as hell”) but many years of Visual Prolog programming dictate the *necessity* of using such “dirty methods” when we are in *desperate need* to process *huge amounts of data* in RAM, rather than in hard disks.

Of course, today's hard disk databases (with SQL, etc) are more efficient than ever before. However, there are *important advantages* in keeping as much as possible of our data in RAM, when we want to do (e.g.) *exhaustive fuzzy string-matching*, cross-examining each list-element versus all other list-elements, etc.

NO HARD-CODED EXTERNAL CALLS

Avoiding *hard-coded external function-calls* was found to be good programming practice.

Today *all* Assembly language predicates in these libraries use *pointers to external calls* instead of *literal* external predicates. So, if (e.g.) a newer version of Visual Prolog (or a new ‘C’ compiler, used for mixed language programming) is installed, it is still possible to use *the same libraries*, with minor changes in *very few* pointer-definition files where the names of some *external predicates* are stored (and are *compiled into pointers* immediately).

Alternatively, it is also possible to use initial calls of certain “configuration predicates”, to fill the locations of external call pointers with addresses of external predicates *at run-time*.

So, instead of **_IO_WrStr** (a built-in Visual Prolog/‘C’ predicate that writes strings to the screen or to current output, defined in the file “pdcrun.h”) many ASM predicates use a call to the *function pointer* “**_wr_str**”, located in the Data Segment. This makes it possible to do *much more* than just writing on the screen: E.g. if the contents of location **_wr_str** are changed *at run-time* to contain the address of **hp_wrstr**, a new ASM predicate that writes strings to “*virtual devices*” inside the *heap*, it

becomes possible to *accumulate* text that may be the result of some processing, writing it *in small lump, inside the heap*, and in the end to dump it -all at once- to a file, or to the screen, *after certain conditions are met*.

Nevertheless, the *default-values* of all these memory locations are *pre-filled* with Visual Prolog predicates, and are *rarely expected to change*: Location “_wr_ch” contains Visual Prolog’s “_IO_WrCh”, location “_wr_int” contains Visual Prolog’s “_IO_WrIntg”, and so on. If any of these need changing, there are appropriate ASM-predicates to do the job.

For example, in order to change *at run-time* the current string-writing predicate, there is a *configuration predicate*, “set_wr_str/1”:

```
GLOBAL DOMAINS
  STR_i: determ (STRING)
        -(i) language c
```

```
GLOBAL PREDICATES
  set_wr_str( STR_i )
        -(i) language c
```

Calling “set_wr_str(foo)” where “foo” is *any* predicate of domain “STR_i”, results in “foo” becoming the *current string-writing predicate* globally called by *every* ASM predicate (that does *any* calls to any *external string-writing*).

Now, “foo” is *not “obliged”* to do any *literal* string-writing; Instead of this, it could very happily be asked to do *other* things, such as calling “assert” (so that -instead of writing- it *asserts facts* in a Prolog database, and so on).

Another common example (part of which was discussed earlier, with “strx2list”) is this: To change at run-time the *current global memory allocation predicate* used (by “strx2list”, as well as others), there is a global *configuration predicate*, “set_allocfunc”:

```
GLOBAL DOMAINS
  allocbin_func
  = determ BINARY (UNSIGNED size)
        -(i) language c
```

```
GLOBAL PREDICATES
  set_allocfunc(allocbin_func)
        -(i) language c
```

The possibility of changing the *currently used memory-allocation predicate* (globally) by a program *on the fly* is also useful when calling these predicates inside a “main program” that is in a language *other than Visual Prolog*.

SPECIALISED WRITING PREDICATES

The ASM-predicate **write_slist**, which writes a list of strings with *separators*, belongs to a large family of similar writing predicates that can have their output *redirected* anywhere by changing the *function pointers* called by them for writing. The default-values of the pointers (as explained before) are set to Visual Prolog writing predicates, but can be changed to *any* other predicates at any time.

The *family* where “write_slist” belongs *also* includes predicates that perform sophisticated *writing scripts*: Some of them can write the contents of Visual Prolog binaries in ways understandable by humans; others may write integer-lists, strings and binaries in ways that are needed by *specific software applications*. (e.g. to create CSV Excel-files).

All these can be *redirected* for writing to the heap (in “virtual strings”), or for doing other (often unexpected) tasks *irrelevant to writing*: *Any* predicate of similar **type** to “_IO_WrStr” may be used as a “virtual writing predicate”, by inserting its address in location “_wr_str”.

In Appendix C there is a complete listing of all these “*special writing predicates*”, with some brief comments about their *operation*.

Appendix C also includes another *big family* of ASM predicates, which are often useful for experimentations and software development, but are *not* really *so* important for most final applications: **Random generation predicates**.

CSV-FILE HANDLING PREDICATES

In Appendix **D** there is a listing and a brief description of an Assembly language library to process CSV-files (Excel-) files, in many *possible ways*. Some of these ASM predicates analyse CSV files to extract parameters from them (e.g. their number of data-fields); others write them out in various ways; others select, delete or insert columns, rows, or cells, etc.

However, due to *space and time limitations* this CSV-file handling library is *not* included in the *main part* of this presentation.

USING THE HEAP FOR “STREAMING”

A common need is to suspend actual writing (to the screen, or to a file), writing everything to a virtual device or “stream”, which can be *flushed*. So, another family of predicates was created for this; writing strings, numbers and characters to “virtual strings” inside the *heap*.

First, the initialisation-predicate “**heap_init**” should be called, with a single argument: the number of virtual *heap strings* required.

This call allocates heap-memory for the *array of pointers* (and all the “position indicators”) needed by the “virtual strings”, but *not* for the strings *themselves*. To allocate heap memory for each *individual virtual string*, predicate **heap_create/1** must be called, once for each “virtual string”: It has only one argument, the *required size of every string*; It returns (in register EAX) the “*integer-handle*” for each “virtual string”. This “handle” can then be re-used for “virtual writing”, i.e. adding content to the specific “virtual string” in ways that are similar to writing on the screen, or to a file.

If we don’t need *many* such “virtual strings”, or if we want to use one of them *most of the time*, a “current default string” can be defined by predicate “**hpstream/1**”, with a single argument: -This virtual string’s “handle”.

Some Assembly language predicates in this “heap-string library” emulate precisely Visual Prolog’s `_IO_WrStr`, `_IO_WrCh`, `_IO_WrIntg`, etc, but (this time) writing to “virtual strings”, rather than to the screen or to a file. Their “target” is the “current virtual string” (which can be changed by “**hpstream/1**”). These are predicates that can be inserted as “clones” of *standard writing predicates*, in the locations invoked by “**write_slist**” (and many others).

See Appendix **E** for a listing of all the *global predicate definitions* in this specialised heap-writing and “streaming” library.

ENHANCED SEARCH PREDICATES

An entirely new family of predicates for searching (inside strings, as well as inside binaries) was implemented in Pure Assembly language. This family of predicates contains many enhancements, including *fuzzy search*, *non-deterministic search*, etc), in order to:

- 1) **Start from a pre-specified position** (rather than always at the beginning)
- 2) **Return the “rest”** (after the search, as a *pointer*; *not* through memory allocation)
- 3) **Search backwards** (*used in parsers*)
- 4) **Search non-deterministically** (a *Prolog-specific* feature, useful for *Data Mining*)
- 5) **Search case-insensitively**, and with **Generalized Case-insensitivity** based on **Character Table techniques**
- 6) **Do fuzzy searches and inexact pattern-matching Searches** (in many ways),
- 7) **Search optimally, using the Boyer-Moore algorithm** (and other types of *optimized search*)
- 8) **Do everything for both Strings and Binaries** (*providing all possible variants of all these predicates*, together with all their *features*; as a result, e.g. there are **15 different variants of “nd_search”** alone, which does *non-deterministic search*)
- 9) **Handle all possible types of objects** (I.e. be able to find strings or binaries, or characters inside strings or binaries, in *all possible type-combinations*).

In Appendix F there is a complete listing of all the *Global Predicate definitions* for this large family of **universal search-predicates**.

Of particular interest are **non-deterministic** and “**fuzzy search**” predicates in this family.

1. All **non-determinism** in these predicates was customized to fit the inner workings of *Visual Prolog*. It is not at all easy to implement non-determinism in *Assembly language*, and it is also **much harder** (or even impossible) to implement it for most other Prolog compilers. At least in Visual Prolog it is guaranteed to remain 100% compatible with non-deterministic code written in Visual Prolog itself, which is nice.
2. **Fuzzy search predicates** have been implemented in a variety of ways, some of them customized for applications. The most common method for these is the use of **Character Tables**, together with repeated calls of the Assembly Language instruction “**XLAT**”: If the *commonest possible erroneous values* of every character are already known, then *weights* can be assigned to these values in a table (with the highest weight assigned to the correct value). Search algorithms can then exploit the *sum of these weights*: -As the text is scanned, each character is “translated” to a value in the table, using instruction “**XLAT**”, this value is accumulated and its accumulated sum is finally compared to a *threshold*.
3. **Inexact string-matching**: Experimentally it was discovered that *the combination* of the above **XLAT**- / *character table-based* techniques with Assembly-variants of the well-known “**Levenstein algorithm**” for *inexact string-matching*, gave the *best* results, and also the best possible manual or automatic *data-specific refinement* of the matching process,

Of course, these simple (sometimes *ad-hoc*) methods of **fuzzy search** may seem somewhat childish to **bio-computing experts**, who know *more efficient algorithms for inexact string matching*, used in today’s **DNA Research**.

However, the *boosting effects* of *Assembly Language* on an algorithm’s *execution speed* often *compensate* for the algorithm’s *minor imperfections*. Besides, it may be worthwhile to *rewrite* today’s *best* bio-computing search algorithms in *hand-optimized Assembly*, and then perhaps use **Visual Prolog** for the more *intelligent parts* of DNA analysis, e.g. using professor *David Searle*’s recently discovered *String Variable Grammar* (already written in Prolog) to delve into *semantic DNA parsing*!

In any case, a combination of character table-based methods with Levenstein’s algorithm was used in a *Visual Prolog ENB application* that correctly identified *the names* of many hundreds of hydrological stations, matching each station’s *true name* to many commonly used aliases and *sub-station names*. The lack of a *common convention* about hydrological station names -in previous years- had caused a *painfully prolific* number of *aliases*, inside paper documents of the *raw* hydrological data for *previous decades*.

In Appendix G there are selected screen-shots from this *Visual Prolog Windows program*, which was used by ENB Ltd to compile and correct the final and official ***Nationwide List of Hydrological Stations in Greece***.

In Appendix G there are also some source code *sample listings*, for both **fuzzy** and **non-deterministic** search predicates (of global definitions provided in Appendix F).

DATA MINING IN BITS (1): “DATE-BINARIES”

“**Date-Binaries**” are data-structures of Dates encoded *as bits*.

A “**Date-Binary**” is a *Visual Prolog Binary* with the following structure:

1. A four-byte header, which contains (a) the number of Years present inside the Date-Binary, encoded in the first 2 bytes, and (b) the Starting Year, encoded in the *next* two bytes. Both these parameters have a range of 0 - 65535, more than adequate.
2. A number of “year-records”, each one containing precisely 12 “month-records”. These month-records are 4-byte (32 bit-) objects, containing days *encoded as bits*.

In addition,

- **Date-Binaries** contain *complete years*. Each year has a size of *48 bytes*, or *12 double words*, each one being *a month*, where *each 31 bits represent the days of the month* and *1 bit* is available as a “flag-bit”, for use by specialized operations.
- **Date-Binaries** have been created to implement a number of *statistical, logical* and *data-mining operations* on time-series data, *in the fastest and most compact way possible*.
- **Date-Binaries** can be combined with *other* forms of data, e.g. hydrological measurements. Such additional data can be numeric arrays, records of various types in SQL databases, GIS maps, EXCEL-file fields/records, etc.
- **If the position of a bit is known**, inside a **date-binary**, then the *date* represented by this *bit* can instantly be decoded, while any corresponding *record* with a certain calendar-index equal to the *bit's position* is also very naturally accessible.

In short, **date-binaries** consist of *bit-patterns* or bit-sets, representing the *days* of specific *months* within specific -series of- *years* in a compact way, allowing *fast and easy access*.

Date-binaries are *complete* and *adequate*, as specifications of *any* single *Boolean attribute*, over calendar-time. This attribute can denote truth or existence of *any* property, of *any* type of data in relation to calendar time.

E.g. date-binaries could be used to analyze the *market behavior* of customers who did or did not buy a certain product, over time, and who did or did not have an “attribute” that we *think* is *relevant*, in the likelihood to buy this product, within given periods of time. In this particular -hypothetical- example, all we need to do, to verify a *suspected latent correlation* between buying patterns of customers who do or don't have a “specified attribute”, is to use *date-binary bit-operations* (AND, OR and NOT) at a *massive scale*, in various ways.

So, Assembly language predicates have been written that implement a variety of logic and numeric **bit-operations on Date-Binaries**:

-Union and Intersection (like Set-operations), deletion, insertion, writing to the screen or to the current output device, bit-operations of statistical nature (or at a slightly higher level) such as *counting the number of bits or days* inside each month (to produce the *density of valid measurements* in a period of years), etc.

Date-binaries were originally created to assist the quest for the *longest intervals of valid and contiguous hydrological time-series data*, in a large-scale project of hydrological predictions financed by the Greek government.

The problem of this project was that much of the raw data, sent from hydrological stations throughout the country, was partly *incomplete* and *unusable*. Soon enough, it had proved to be *far more tedious* to distinguish the *good* from the *bad* data, than to actually *use it!*

The company's main hydrological software used by ENB's engineers at the final phase became *useless* if the data fed to it had gaps of *erroneous data*, inside the *raw time-series*.

So, a Visual Prolog *Windows application* was developed, based on date-binaries, to produce reliable data-summaries, readable by humans, for use in *important decisions on this project*.

To identify the *erroneous data-gaps*, a *five-phase extraction process* was devised:

1. First, the *dates* of all the time-series were turned into bits / date-binaries and *invalid raw data* was excluded.
2. Then, the date-binaries were AND-ed together, finding *common days* (or bits)
3. Then, all valid common days inside each month, or *bits* in each *double word*, were counted and compared to a *threshold of 25 days per month*, producing *common valid month – binaries*, where each month was encoded as either 1 (valid) or 0.
4. Then a special Assembly language predicate was used to find the sizes of all contiguous time-periods, or *consecutive 1's*, inside *these common month binaries*
5. Finally, a (Visual Prolog) *fail-driven loop* was used to scan through the resulting *sizes* (of *contiguous time periods*) finding their *maximums* for *all possible pairs of time-series data*. These maximums were then plotted in CSV-file format and transferred to the engineers of ENB Ltd.

The use of date-binaries in this project was a *clear success*. All known previous methods for achieving similar goals became *obsolete*. *Visual Prolog* program development was also fast: *Less than a day*, plus about ten working person-days to... invent the “Date-Binaries Library” *itself*; *i.e.* the ASM predicates used by the program. This program was *so fast* that it took longer to save results to a file than to... calculate them, even though the data included *several decades* and *many dozens of stations*.

Admittedly, it would be possible to use *any* “mainstream language”, such as Visual Basic or C++, to achieve the same goals. However, even forgetting the advantage of a *very short development time* (using *Visual Prolog*) the ultra-high *execution speed* of the program empowered the company’s engineers to

experiment widely with many different ranges, settings, thresholds and time-periods, so as to re-adjust and re-define the *complete picture of the data*. This had a very positive impact on the *quality of the decisions* taken, about *which parts* of the raw data to accept.

There are *screen shots* of this Visual Prolog application of date-binaries, in Appendix **H**.

We now have *good reason* to believe, that the software methodology used for assisting this project, can achieve similarly good results in *any* field where *incomplete* or *erroneous* data is a *serious obstacle*: -Which means: *Nearly every other company, similar to ENB Ltd, throughout the world (working with raw time-series data)*. However, the feeling of the author is that *this is only the beginning*. A logical next step is to associate *date-binaries* with other types of data and *rule extraction algorithms*, e.g. operating on raw time-series to discover hidden patterns or hidden relations inside the data, at a very high speed. Now, “Inductive Logic Programming” methods are known to demand vast amounts of time to arrive at the simplest results, but if they are assisted or boosted with such low-level innovations (as date-binaries), they are bound to become much faster and much more practical, in business fields that are extremely demanding for *instant and powerful results*.

In Appendix **H** there is also a listing of all *Global Predicate definitions* in our libraries that deal with Date-Binaries.

In the next section, we will take a *brief look* into a methodology *much more powerful* and important than “date-binaries”, but also 100% compatible with them: It's a methodology that can assist serious and extremely reliable data-mining processes *at a fraction of the time* normally required for them: A *logic inference algorithm* which (like date-binaries) *also* uses bits and is also based on *low-level logic bit-operations*, but it is of *more general scope*, a part of an “Alternative Logic System” which could be described as a “*Reduced Instruction Set Inference Engine for Boolean Logic*”:

DATA MINING IN BITS (2): “ALGORITHM IPHIGENIA”

The “Iphigenia algorithm” is a certain kind of “mathematical curiosity”, or invention of the author, dated back to the mid-eighties. Today, after many years -during which the *theory* of this algorithm was... buried in the closet- there are fresh results and theorems proved in 2004. Assembly Language implementations of this algorithm also led to some advances in the theory behind it: “**Multiple Form Logic**”, about which you can find out more in the site <http://multiforms.netfirms.com> (a strictly *non-commercial* research site).

Unfortunately, “**Multiple Form Logic**” is *not* easy to expound *in just a few minutes*, and it is also considerably *off topic* for the concerns of this conference. So this particular section will come to an end with a brief discussion of practical results obtained by using algorithm Iphigenia and also some *hints* about how this algorithm can be combined with certain other *compatible methodologies*, implemented in Assembly language, such as *date-binaries*, in the context of *Visual Prolog programming*.

One of the *easiest* results to deduce (at a *very* high speed of execution), using the Iphigenia Algorithm implemented in Assembler, is the fast checking of whether or not a collection of logic facts of the form “A and B and C and... => X or Y or...” when combined with a new question or a “logic query”, is “*true*” or not.

The algorithm operates blindly and with brute force, as a bit-pattern reduction and inference engine, in a way that is “pseudo-parallel”, and it *crunches our bit-encoded knowledge base* in a *fraction of a second*, to find the result.

There only very few “bit reduction rules”, all of which are implemented as *bit operations* on Visual Prolog *binaries*. There are a couple of rules for “vertical or horizontal” reduction, a couple of “row elimination rules”, and... *well, that's it !*

If our Knowledge Base is *encoded as bits*, it can also be easily *extended* to include *date-binaries*, or *any other bit-encoded attributes* (of *raw unexplored information*). Algorithm Iphigenia can then also be used as a *powerful data-mining engine*, able to deduce new logic patterns and inferences (*in bits*) that were *not known* (at the start), provided we make sure that the *interface* between the “raw data” and our *logical “meta-rules”* (used for the data mining *strategy*) is kept *transparently clear* and *well-designed*. Quite often it is necessary to include *external criteria* or some types of “thresholds” at an intermediate level, *before* “raw data” is allowed to pass into the “blind” *logic inference phases*. E.g. in the previously mentioned hydrological project -at ENB Ltd- where we used *date-binaries*, *without* using algorithm Iphigenia, a *threshold* of 25 days per month was decided to be the “minimum number of days for a particular month's data to become acceptable” by *subsequent phases*.

A common mistake that one can make easily, if one uses this algorithm *carelessly* is to *ask a wrong question*, causing an “avalanche” of useless but... *correct* inferences, which -even though *are* (logically) correct- they are also... *impossible to digest* (as regards their results) unless you possess... supernatural abilities for *logical complexity comprehension*. The larger our knowledge base, the more logic facts *will* be deduced massively and blindly (but also... *correctly*) as *logic consequences* of our query. If our question is reasonable, then the answer will *also* be reasonable. If -however- our query is *ambiguous*, then the answers are *not* produced by anything... decent, polite and sequential as (Prolog-like) back-tracking, but -instead- they are *hammered out* by a brute-force *potentially parallel* inhuman algorithm of *blind reduction* at lightning-fast speed, but also *hard to stop* (or control), *once it begins!*

In Appendix **I**, recent results from *running* Algorithm Iphigenia are given, with selected *screen shots* of the *testing programs used*.

This page is intentionally left blank

APPENDIX A: Sample source-code in Assembly Language.

```

; ===== _strx2list.asm
; =====
; GLOBAL PREDICATES
; SLIST strx2list (STRING tx,CHAR sepc) -(i,i) language c
; SLIST strx2list (STRING tx,CHAR sepc,BINARY buffer) -(i,i,i) language c
;
; Purpose:
; Ultra-fast and memory-optimised conversion of a (separator-)delimited string
; (ARG 1) to a string-list (returned in EAX). The separator is placed in ARG 2.
; Notes:
; These predicates do not allocate memory for any strings. They "re-use" the
; input-string after "decimating it" by replacing every separator-char (ARG2)
; with an ASCII 0, while placing pointers to the emerging pieces inside the
; list. Moreover, only the first predicate actually allocates memory, for the
; list itself (where each list-element requires 12 bytes). The 2nd predicate
; does not allocate any memory; it uses a "binary buffer" instead (ARG 3),
; which must be created in memory (and be of adequate size) before using it.
; In the first predicate, memory is allocated once-and-for-all for the entire
; output-list; NOT for each list-element. Contiguous memory is used for the
; list-elements, to keep their generation simple, fast and elegant.

        IDEAL
        P586
MODEL   FLAT
        DATASEG
ALIGN   4
        extrn _allocfunc:dword
vdblist dd vdb
vdb dd 2,0,0
        CODESEG
ALIGN   4
public  _strx2list_0      ; main predicate, for use by PROLOG and 'C'
public  _strx2list_1      ; main predicate, for use by PROLOG and 'C'

PROC    _strx2list_0 near ; -(i,i)
ARG     strg:dword, sepc:byte
        push  ebp
        mov   ebp,esp
        push  esi
        push  edi
        push  ebx
        mov   ecx,[strg] ; ARG 1 = string
        mov   al,[ecx]   ;
        or    al,al      ;
        jz   short @@VDx ;
; -----
        mov   ah,[sepc] ;
        mov   edx,1      ;
        xor   ebx,ebx     ;
        jmp  short @@C1  ;
; =====
@@C0:   mov   [ecx],bl    ; replace sepc with zero
        inc   edx        ;
        inc   ecx        ;
@@C1:   mov   al,[ecx]   ;
        cmp   al,ah     ;
        jz   short @@C0 ;
; -----
        inc   ecx        ;
        or    al,al     ;

```

```

        jnz short @@C1      ;
; ----- now EDX = number of list-elements
        push ecx           ; save the end of the input-string
        inc  edx           ; one more element (for void end-of-list)
        shl  edx,1         ;
        mov  eax,edx       ; now EAX = EDX = 2 * num_elements
        shl  eax,1         ; now EAX = 4 * num_elements
        add  eax,edx       ; now EAX = 6 * num_elements
        shl  eax,1         ; now EAX = 12 * num_elements
        push eax           ;
        call [_allocfunc] ;
        pop  ecx           ;
        pop  ecx           ; now ECX = end_of_string
        mov  esi,eax       ; newly-allocated list in ESI
        push eax           ; save it also in the stack...
        mov  eax,[strg]    ;
        sub  ecx,eax       ; now ECX = string_length+1
        dec  ecx           ;
        mov  edi,eax       ; now EDI = ptr_to_input_string
        mov  ebx,1         ;
        mov  [esi],ebx     ;
        mov  [esi+4],edi   ; first element = start_of_string
        mov  edx,12        ; use EDX = 12 as a constant
        add  esi,edx       ;
        mov  [esi-4],esi   ;
        xor  al,al         ; use AL = 0 for searching
@@L1:   repne scasb        ; search for a zero-character in EDI
        jecxz @@x1        ;
; -----
        mov  [esi],ebx     ; write a '1'-flag for normal list-element
        mov  [esi+4],edi   ; place the start of next_string in the list
        add  esi,edx       ;
        mov  [esi-4],esi   ;
; -----
        jmp short @@L1     ; repeat till end_of_string
; =====
@@x1:   mov  eax,2         ;
        mov  [esi],eax     ; write 'last element flag'
        xor  eax,eax       ;
        mov  [esi+4],eax   ;
        mov  [esi+8],eax   ; created void list-element (at the end)
        pop  eax           ; recover start of new list, return it in EAX
        pop  ebx           ;
        pop  edi           ;
        pop  esi           ;
        pop  ebp           ;
        ret
; *****
@@VDx:  ; case of empty_string --> empty_list
        mov  eax,[vdlist] ; return empty list (pre-defined in memory)
        pop  ebx           ;
        pop  edi           ;
        pop  esi           ;
        pop  ebp           ;
        ret
ENDP   _strx2list_0

PROC   _strx2list_1 near   ; -(i,i,i)
ARG    strg:dword, sepch:byte, list:dword
        push ebp
        mov  ebp,esp
        push esi
        push edi

```

```

    push    ebx
    mov     ecx,[strg]    ; ARG 1 = string
    mov     al,[ecx]     ;
    or     al,al        ;
    jz     short @@VDx   ;
; -----
    mov     ebx,12      ;
    mov     edi,[list]   ; pre-allocated list in EDI
    mov     esi,1       ;
    mov     [edi],esi    ;
    mov     [edi+4],ecx  ; first element = start_of_string
    add     edi,ebx     ;
    mov     [edi-4],edi  ;
    mov     ah,[sepch]   ; ARG 2 = sepchar
    jmp     short @@L1   ;
; =====
@@L0: mov [ecx],bh      ; put a zero in place of the sepChar
    inc   edx          ; add 1 to the elements_counter
    inc   ecx          ; advance str-pointer
    mov   [edi],esi    ; write a '1'-flag for normal list-element
    mov   [edi+4],ecx  ; place the start of next_string in the list
    add   edi,ebx     ;
    mov   [edi-4],edi  ;
@@L1: mov al,[ecx]     ; read a char
    cmp  al,ah        ; is it a separator?
    jz  short @@L0    ; if so, goto special processing
; ----- else...
    inc  ecx          ; advance str-pointer
    or   al,al        ; was the char a zero?
    jnz short @@L1    ; if not, repeat till end_of_string
; ----- else (end of string)
@@x1: mov  eax,2      ;
    mov   [edi],eax   ; write 'last element flag' (2)
    xor   eax,eax     ;
    mov   [edi+4],eax ;
    mov   [edi+8],eax ; created void list-element (at the end)
@@xx: mov  eax,[list] ; return the start of the list in EAX
    pop   ebx
    pop   edi
    pop   esi
    pop   ebp
    ret
; *****
@@VDx: mov  eax,[vdlist] ; case of empty_string --> empty_list
    pop   ebx
    pop   edi
    pop   esi
    pop   ebp
    ret
ENDP   _strx2list_1

    END

; ===== VISUAL PROLOG TEST-PROGRAM: =====
GLOBAL PREDICATES
    SLIST strx2list (STRING tx,CHAR sepc) -(i,i) language c
    SLIST strx2list (STRING tx,CHAR sepc,BINARY buffer)
        -(i,i,i) language c

PREDICATES
    test (INTEGER)
    dump2file (CHAR,UNSIGNED,STRING,SLIST)

```

```

CLAUSES
test(0):- mrandominit(723), set_allocfunc(alloc_heapbin), repeat,
  storage(_,HEAPsz,_), write("\nHeap memory available: ",HEAPsz),
  write("\nGive size of input-text (in the heap) to be randomly generated:\n?- "),
  readint(SZ), write("OK. Now calling 'alloc_heapstr/1'..."), S =
alloc_heapstr(SZ),
write("\nDONE! (created the string in Visual Prolog's heap)..."),
write("\nNow calling: fill_randstr_bytes(STRING,\"SZ,\"A','Z')...\n"),
fill_randstr_bytes(S,SZ,'A','Z'),
write("\nDONE! (Filled the string with random characters 'A' to 'Z')."),
write("\nGive the number of list-elements (or text-newlines): "), readint(Nlin),
write("\nOK. Now creating \"Nlin,\" newlines..."), Wid = SZ div Nlin,
MaxPx = Nlin-1, getbacktrack(LP1), repinc(1,MaxPx,NN), Posx = NN*Wid,
nth_byte(S,Posx,'\n'), NN >= MaxPx,
cutbacktrack(LP1), Nc1 = count_chars(S,'\n'), Nc=Nc1+1,
write("\nDONE! The random text now has \"Nc,\" newlines.\"),% nl, write(S),
write("\n\nPress a key to run strx2list/2 -(i,i):"), readchar(_),
time(Hours1,Mins1,Secs1,Hunds1), SLx = strx2list(S,'\n'),
time(Hours2,Mins2,Secs2,Hunds2),
TMx = timedif(Hours1,Mins1,Secs1,Hunds1,Hours2,Mins2,Secs2,Hunds2), TMxx = TMx *
10,
writef("OK\nDONE! The text was converted to a list, in: %",TMxx),
write(" milliseconds!\n",
  "Press SPACE to dump the list to a file, ESC to stop, else continue:"),
readchar(CHx), dump2file(CHx,Nc,"test1",SLx),
write("\nNow calling 'release_heapdw/2'..."), DW = cast(dword,SLx),
release_heapdw(DW),
write("\nFinally calling 'release_heapstr/2'..."), release_heapstr(S,SZ),
write("\nReleased all temporary memory used.\n"), searchchar("\27 ",CHx,_), !.

test(1):- mrandominit(723), set_allocfunc(allocgstack), repeat,
  storage(_,HEAPsz,_), write("\nHeap memory available: ",HEAPsz), MARK =
markstack(),
write("\nGive size of input-text (in the heap) to be randomly generated:\n?- "),
readint(SZ), write("OK. Now calling 'alloc_heapstr/1'..."), S =
alloc_heapstr(SZ),
write("\nDONE! (created the string in Visual Prolog's heap)..."),
write("\nNow calling: fill_randstr_bytes(STRING,\"SZ,\"A','Z')...\n"),
fill_randstr_bytes(S,SZ,'A','Z'),
write("\nDONE! (Filled the string with random characters 'A' to 'Z')."),
write("\nGive the number of list-elements (or text-newlines): "), readint(Nlin),
write("\nOK. Now creating \"Nlin,\" newlines..."), Wid = SZ div Nlin,
MaxPx = Nlin-1, getbacktrack(LP1), repinc(1,MaxPx,NN), Posx = NN*Wid,
nth_byte(S,Posx,'\n'), NN >= MaxPx,
cutbacktrack(LP1), Nc1 = count_chars(S,'\n'), Nc=Nc1+1,
write("\nDONE! The random text now has \"Nc,\" newlines.\"),% nl, write(S),
write("      "\n\nPress a key to run strx2list/2 -(i,i):"), readchar(_),
time(Hours1,Mins1,Secs1,Hunds1), SLx = strx2list(S,'\n'),
time(Hours2,Mins2,Secs2,Hunds2),
TMx = timedif(Hours1,Mins1,Secs1,Hunds1,Hours2,Mins2,Secs2,Hunds2),
TMxx = TMx * 10, writef("OK\nDONE! The text was converted to a list, in:
%",TMxx),
write(" milliseconds!\n",
  "Press SPACE to dump the list to a file, ESC to stop, else continue:"),
readchar(CHx), dump2file(CHx,Nc,"test1",SLx),
write("\nNow calling 'release_heapstr/2'..."), release_heapstr(S,SZ),
releasestack(MARK),
write("\nReleased all temporary memory used.\n"), searchchar("\27 ",CHx,_), !.

test(2):- mrandominit(723), repeat, storage(_,HEAPsz,_),
write("\nHeap memory available: ",HEAPsz),
write("\nGive size of input-text (in the heap) to be randomly generated:\n?- "),

```

```

readint(SZ),
write("OK. Now calling 'alloc_heapstr/1'..."), S = alloc_heapstr(SZ),
write("\nDONE! (created the string in Visual Prolog's heap)..."),
write("\nNow calling: fill_randstr_bytes(String,\"SZ,\"A','Z')...\n"),
fill_randstr_bytes(S,SZ,'A','Z'),
write("\nDONE! (Filled the string with random characters 'A' to 'Z')."),
write("\nGive the number of list-elements (or text-newlines): "), readint(Nlin),
write("\nOK. Now creating ",Nlin," newlines..."), Wid = SZ div Nlin, MaxPx =
Nlin-1,
getbacktrack(LP1), repinc(1,MaxPx,NN), Posx = NN*Wid, nth_byte(S,Posx,'\n'),
NN >= MaxPx, cutbacktrack(LP1), Ncl = count_chars(S,'\n'), Nc=Ncl+1,
write("\nDONE! The random text now has ",Nc," newlines."), SZ2 = (Nc+1)*12,
write("\nNow calling 'alloc_heapbin/1'..."), Bin = alloc_heapbin(SZ2),
write("\nDONE! Allocated a temporary 'binary buffer' of size: ",SZ2),
write(".\n\nPress a key to run strx2list/3 -(i,i,i):"), readchar(_),
time(Hours1,Mins1,Secs1,Hunds1), SLx = strx2list(S,'\n',Bin),
time(Hours2,Mins2,Secs2,Hunds2),
TMx = timedif(Hours1,Mins1,Secs1,Hunds1,Hours2,Mins2,Secs2,Hunds2),
TMxx = TMx * 10,
writef("OK\nDONE! The text was converted to a list, in: %",TMxx),
write(" milliseconds!\n",
    " Press SPACE to dump the list to a file, ESC to stop, else continue:"),
readchar(CHx), dump2file(CHx,Nc,"test2",SLx),
write("\nNow calling 'release_heapstr/2'..."), release_heapstr(S,SZ),
write("\nReleased string-memory. Now calling 'release_heapbin/2'...\n"),
release_heapbin(Bin,SZ2),
write("\nReleased all temporary memory used.\n"), searchchar("\27 ",CHx,_), !.

test(3):- !.

dump2file(' ',Nc,NAME,SLx):- syspath(PA,_),
format(Fx,"%_ _ _lines.txt",PA,NAME,Nc),
write("OK.\nDumping the list to file:\n\"",Fx,"\""),
openout(Fx), write_slist(SLx,'\n'), closefx,
write("\nOK. Wrote the file using 'write_slist(SLIST,'\n')'."), !.
dump2file(,,,_):- !.

GOAL
easywin_CloseAfterEnd(b_True), lctr_set(2,1), repeat, INI = lctr_get(2),
dlg_ListSelect("testing 'strx2list'",["str2xlist(String,CHAR)-(i,i) using HEAP",
    "str2xlist(String,CHAR)-(i,i) using GSTACK",
    "str2xlist(String,CHAR,BINARY) -(i,i,i)",
    "EXIT"],INI,SELx,NDX),
write(NDX," ",SELx), nl, lctr_set(2,NDX), test(NDX), NDX=3, !.

```

APPENDIX B: Sample “glue-code” and *global function pointers*

```
; ===== source-file: _wr_funcs.asm =====
; GLOBAL DOMAINS
; str_i = determ (STRING) -(i) language c
; char_i = determ (CHAR) -(i) language c
; int_i = determ (INTEGER) -(i) language c
;
; GLOBAL PREDICATES
; set_wr_str(str_i) -(i) language c
; set_wr_ch(char_i -(i) language c
; set_wr_int(int_i) -(i) language c
;
IDEAL
P586

MODEL flat

CODESEG

ALIGN 4

extrn _IO_WrStr:near
extrn _IO_WrCh:near
extrn _IO_WrIntg:near
extrn _MEM_AllocGStack:near

DATASEG
ALIGN 4

public _wr_str
public _wr_ch
public _wr_int
public _allocfunc

_wr_str dd _IO_WrStr
_wr_ch dd _IO_WrCh
_wr_int dd _IO_WrIntg
_allocfunc dd _MEM_AllocGStack

CODESEG
ALIGN 4

public _set_wr_str
public _set_wr_ch
public _set_wr_int
public _set_allocfunc

PROC _set_wr_str near
ARG pred:dword
push ebp
mov ebp,esp
mov eax,[pred]
mov [_wr_str],eax
pop ebp
ret
ENDP _set_wr_str
```

```

PROC _set_wr_ch near
ARG  pred:dword
    push  ebp
    mov   ebp,esp
    mov   eax,[pred]
    mov   [_wr_ch],eax
    pop   ebp
    ret
ENDP _set_wr_ch

PROC _set_wr_int near
ARG  pred:dword
    push  ebp
    mov   ebp,esp
    mov   eax,[pred]
    mov   [_wr_int],eax
    pop   ebp
    ret
ENDP _set_wr_int

PROC _set_allocfunc near
ARG  pred:dword
    push  ebp
    mov   ebp,esp
    mov   eax,[pred]
    mov   [_allocfunc],eax
    pop   ebp
    ret
ENDP _set_allocfunc

    END

```

APPENDIX C: A library of “virtual heap-string” predicates

Description:

This library is based on an array of pointers to *strings, allocated in Visual Prolog’s heap*. Every such “heap-string” (or “virtual string”) is accessible for “writing”, exactly like an output-device (or a “stream”). The number of these strings is *not* restricted (except by available RAM). An initialisation-call of “heap_init” reserves space in the heap for *any* number of such heap-strings. There is also an internal array of “heap-string positions”, used for writing as well as for remembering the “current position”, inside every “heap-string”. Just like the heap-strings themselves, *both* the array of pointers (to these strings) *and* their “position-array” are *also* allocated inside the heap (dynamically created each time at any size) so that the memory occupied by them can *also* be released totally, by a simple call of “heap_release”.

There are predicates in this library to write *almost anything* to *any* heap-string, and there are other predicates to access *any* heap-string in a *variety of ways*; for example, it is

possible to access the heap-strings “*as if* they are binaries”, or to write inside these strings the contents of Visual Prolog binaries (which may also contain zero-bytes, like ‘C’-strings).

This library was created with implementation of *streaming* in mind. However, its latent capabilities exceed the idea. E.g. recently a new predicate was added, which allows an effective way of “transposing” comma-delimited (or CSV-files), so that (when viewed in a spreadsheet, like Excel) *the rows become columns* and the *columns become rows* (predicate “heap_wr_transposed/2”).

A *new programming methodology* also arose: -The use of fail-driven loops, that “write virtually” (to heap-strings) pieces of information that are subsequently dumped *in their totality* somewhere else (e.g. a text-file). In this way, many types of text or code-listings can be generated, more easily than before (e.g. HTML).

```
UNSIGNED heap_init(UNSIGNED num_heap_strings) -(i) language c
% returns the total memory allocated; needs "heap_create" afterwards

UNSIGNED heap_init(UNSIGNED num_heap_strings,UNSIGNED length_of_each)
    -(i,i) language c
/* Returns memory allocated; does not need "heap_create". It assumes that all the
"virtual strings" are of equal length (equal to ARG 2). */

UNSIGNED heap_release() language c
% frees all heap-memory used; returns the total memory released

heap_reinit(UNSIGNED n_str) -(i) language c
% exactly like "heap_init/1" but calls 'heap_release' first

heap_reinit(UNSIGNED n_str,UNSIGNED len) -(i,i) language c
% exactly like "heap_init/1" but calls 'heap_release' first

INTEGER heap_create(UNSIGNED len) -(i) language c
/* Allocates memory of size = ARG 1 for the "next available heap-string",
returning this string's "heap-handle". */

heap_create(UNSIGNED which,UNSIGNED len) -(i,i) language c
/* Allocates memory of size = ARG 2, for the heap-string of handle = ARG 1*/

INTEGER heap_wrch(INTEGER which,CHAR ch) -(i,i) language c
/* writes a character (ARG 2) to the heap-string of handle = ARG 1 */
```

```

INTEGER heap_wrch(INTEGER which,CHAR ch,UNSIGNED howmany) -(i,i,i) language c
/* writes a character (ARG 2) to the heap-string of handle = ARG 1, a given
number of times (ARG 3) */

INTEGER heap_write(INTEGER which,STRING str) -(i,i) language c
/* writes a string (ARG 2) to the heap-string of handle = ARG 1 */

INTEGER heap_write(INTEGER which,UNSIGNED int) -(i,i) language c
/* writes an integer (ARG 2) to the heap-string of handle = ARG 1 */

INTEGER heap_write(INTEGER which,BINARY binaryblock) -(i,i) language c
/* writes a binary (ARG 2) to the heap-string of handle = ARG 1 */

INTEGER heap_write(INTEGER which,STRING str,UNSIGNED len) -(i,i,i) language c
/* writes a prefix of length=ARG 3 of a string(ARG 2) in the heap-string of
handle = ARG 1 */

STRING heap_get(INTEGER which) -(i) language c
/* returns the heap-string of a given handle (ARG 1) */

STRING heap_dump(INTEGER which) -(i) language c
/* returns the heap-string of a given handle (ARG 1), and also frees the memory
allocated for it, exactly like a call of 'heap-del" */

BINARY heap_bdump(INTEGER which) -(i) language c
/* returns as a binary the heap-string of a given handle (ARG 1), and also frees
the memory allocated for it, exactly like a call of 'heap-del" */

heap_del(INTEGER which) -(i) language c
/* deletes a heap-string of handle=ARG 1, freeing the memory allocated for it */

STRING heap_setpos(INTEGER which,UNSIGNED pos) -(i,i) language c
% sets the 'current position" of a heap-string of handle=ARG 1, to a value=ARG 2.

INTEGER heap_listall() language c
/* gives an ASCII-text complete listing of all the heap-strings, with some
additional information useful for program development and debugging. */

UNSIGNED heap_wr_transposed(CHAR sepc,CHAR newsepc) -(i,i) language c
/* dumps (to Visual Prolog's current output-device) all the contents of every
heap-string, but in a way useful for "transposing a matrix": Assuming all the
heap-strings are delimited with a separator-char (ARG 1), every call of this
predicate collects the "first values" from all the active heap-strings (in
ascending handle-order), writes them out to the output delimited by a "new
separator" (ARG 2) and then adjusts the internal heap-position pointers to get
ready for writing the "next values" from every heapstring. If the separators are
newlines, Repeated calls of this predicate (followed by a newlines) produce a
text-file where line 1 consists of the first lines of every heap-string, line 2
consists of the second lines of them, etc. */

hp_wrstr: STR_i /* writes a string inside the "currently chosen heap-string";
compatible with Visual Prolog's "_IO_WrStr", it can be used to "redirect" the
writing action of many other ASM predicates */

hp_wrch: CHAR_i /* writes a character inside the "currently chosen heap-string";
compatible with Visual Prolog's "_IO_WrCh", it can be used to "redirect" the
writing action of many other ASM predicates */

hp_wrint: INT_i
/* writes an integer inside the "currently chosen heap-string"; compatible with
Visual Prolog's "_IO_WrIntg", it can be used to "redirect" the writing action of
many other ASM predicates */

```