

An Introduction for Designers and Contributors to the GNU APL Interpreter

Dr. Jürgen Sauermann, GNU APL

Table of Contents

Abstract

This document contains a number of unrelated hints that may be useful to understanding the GNU APL source code. Its primary purpose is to support the fading memory of the author, but it may be interesting for others as well. In addition to the hints there are also a number of rules. These rules should be observed by those who contribute C/C++ source code to the GNU APL project (as opposed to those that contribute APL code).

This document is **NOT** aimed at the normal GNU APL user, i.e. at the APL programmer. Those users are referred to the various **README** files in the top-level GNU APL directory.

The GNU APL Build System

All commands described below shall be executed in the top-level GNU APL directory. The top-level GNU APL directory is the directory that contains - after unpacking a GNU APL tar file like **apl-1.9.tar.gz** or after fetching the GNU APL sourced with SVN or with git - the file named **configure.ac**.

GNU APL uses **automake** and **autoconf** to build the interpreter. The build system performs a number of major steps in typically different places and by different persons (C/C++ designer/maintainer vs. user):

Step 1: Generate ./configure and a Makefile.in for every Makefile.am

Every directory of the GNU APL source tree contains a file named **Makefile.am**. The command **autoreconf**, performed in the top-level directory of the GNU APL source tree:

```
$ autoreconf
```

Command **autoreconf** generates a file named **Makefile.in** for every **Makefile.am**. In addition it creates a script named **configure**. This is normally done by the maintainer and the **Makefile.in** files produced are shipped with GNU APL (tar file or svn/git repository).

The absolutely cool thing about **autoconf** (as opposed to e.g. **Cmake** or GUI based IDEs) is that **autoconf** and **automake** themselves need not be installed by the end

users. The **configure** script produced by **autoreconf** is self-contained and can be executed by the end user to configure different aspects of the GNU APL interpreter. To accommodate this, the different **Makefile.in** files are shipped with GNU APL even though they are somewhat redundant (and, compared to the **Makefile.am**, pretty large).

In essence **Makefile.am** files are easy to read and maintain by humans, while **Makefile.in*s are easy to decode by machines, in particular by the *configure** script that is produced along with them. In GNU APL the **autoreconf** command produces a more than 1,000 line top-level **Makefile.in** from a 150 line **Makefile.am** and a 25,000 line `./configure` script from its 1,000 line **configure.ac**



If a maintainer adds a new directory to the source tree, then she needs to add the final **Makefile** in the top-level file **configure.ac** (in m4 macro **AC_CONFIG_FILES**), otherwise the ***Makefile.am** will be ignored.

The most important take-away here is that the user or maintainer should never edit **Makefiles** or **Makefile.ins** because doing so is cumbersome, error-prone, and useless since **autoreconf** will overwrite the changes made.

Step 2: Generate a Makefile for every Makefile.in

The next step is to run the **configure** script produced in the previous step. It is performed by the end user (or by the system administrator where appropriate) in the top-level directory of the GNU APL source tree:

```
$ ./configure
```

The typical end user will run **./configure** without any arguments, while experts may run it with additional arguments (see README-2-configure for details). The primary task of **./configure** is to convert every **Makefile.in** into a corresponding **Makefile**. In doing so, the script tries to determine which and where the libraries and header files used by GNU APL are installed on the user's system. The system functions and variables of the final GNU APL interpreter behaves differently depending on the libraries and header files present. For example, **□FFT** (Fast Fourier Transforms) depends on library **libfftw3** and if that library (or its header files) are missing then **□FFT** will raise a syntax error instead of computing the FFT.



On typical Debian systems, libraries and header files are often contained in different packages that should all be installed.

For example, in the **□FFT** case there are 3 packages:

- **libfftw3-bin** # the library itself
- **libfftw3-dev** # header files for the library, and
- **libfftw3-doc** # documentation related to the library

For **□FFT** to work the first two are needed and the third does not hurt. Most of the other libraries follow the same naming scheme for Debian packages.

The `configure` script produces a lot of output that is useful if a library or its header

file(s) are missing. The final result of the decisions made by configure is collected in file **config.h** so that the compiler can adapt itself to the presence or absence of libraries and their header files.

If a library is not detected even though it is installed then either:

- the test performed by `./configure` is faulty and then either `configure.ac` or else a more complex test file in sub-directory **m4** need to be fixed, or
- the library only lives in a non-standard location (e.g. below the user's home and then `./configure` hopefully provides `./configure` option to specify that location.

Step 3: Compile and Link the GNU APL Interpreter

The final step is to compile the interpreter:

```
$ make
```

The code base is quite large, therefore one may prefer to compile several sources in parallel. For example, on an 8 core CPU:

```
make -j 7
```

leaving one core for other purposes. If the compilation is successful then the interpreter is named **apl** in directory **src**, i.e. **src/apl**. During development, the interpreter can be started from there without installing it:

```
$ src/apl
```

After the development is done, the interpreter is installed properly.



The steps described only require write permission in and below the top-level directory, for instance below the user's \$HOME directory. The next step requires write permissions in directories that are normally not writable by ordinary users.

Step 4: Install the GNU APL Interpreter

Installation of the interpreter must be performed by root:

```
$ sudo make install
```



A common mistake (by the author) is to try to install the interpreter after the previous step has failed. The build system will then try to recompile missing files, but the resulting files will now be owned by root and will no longer be writable by the user. To fix this:

```
$ sudo chown -R username.username . # assume user is 'username'
```

The **make install** installs a few files (all owned by root, but with read permissions

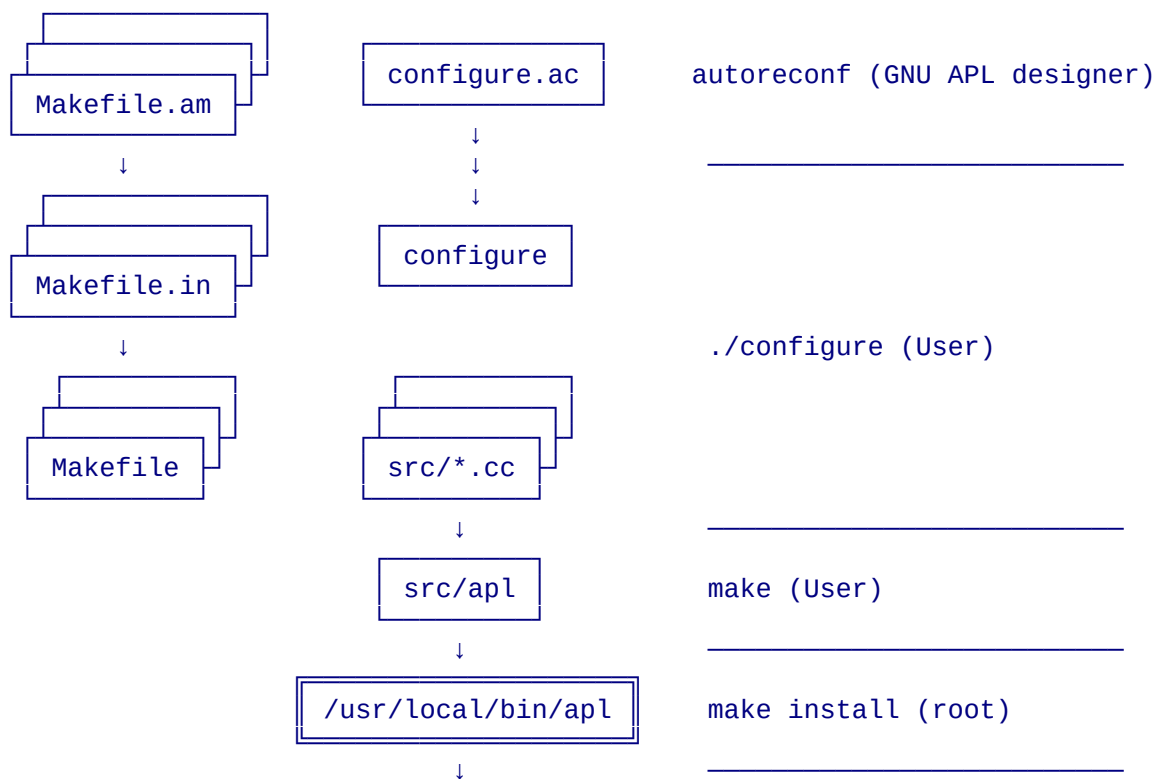
for everybody):

- The APL interpreter itself in **/usr/local/bin/apl**,
- interpreter-related header files **/usr/local/include/apl**,
- interpreter-related libraries **/usr/local/lib/apl**,
- an info file in **/usr/local/share/info/apl.info**,
- a man page in **/usr/local/share/man/man1/apl.1**

The default installation prefix **/usr/local** can be changed with **./configure** options.

Summary

The following picture illustrates the different steps to build the GNU APL interpreter and the files involved:



Special make targets

The typical user shall simply run **make** in the top-level directory.

For experts there are some more make targets (defined in **Makefile.incl**), again with **make** run in the top-level directory, unless noted otherwise.

make help

Shows all top-level make targets:

In addition to the standard make targets (all, install, clean, ...), the following make targets may be supported (at this level):

make help	- print this text
make DIST	- dist + a Makefile that calls ./configure
make DOXY	- create Doxygen documentation
make RPM	- create (source and binary) RPMs
make DEB	- create Debian packages
make SVNUP	- update from SVN and ./configure

NOTE: The RPM and DEB targets may fail because they require additional tools that may not be present on your machine. Don't worry if that happens, unless you really need the RPM and/or Debian packages.

The following targets are shortcuts for lazy developers (like the GNU APL author) and are not very useful for normal users:

make develop	- enable full dependency tracking
make gprof	- make develop + enable gprof profiling
make python	- make develop + build python module
make parallel	- enable multi-core APL (buggy and experimental!)
make parallel1	- make parallel for benchmarking
make VPATH_clean	- prepare for VPATH build

make test

This make target is only available in sub-directory **src**. It builds the interpreter and then runs all testcases (= files with extension **.tc**) that are present in directory **src/testcases**. The total result of all testcases is stored in **src/testcases/summary.log**, and every testcase **x** produces a log file **src/testcases/x.tc.log** with details of the test execution.

make apl.lines

This make target is only available in sub-directory **src**. It produces a file named **src/apl.lines** and must be run right after building the interpreter. When the interpreter produces a stack trace (i.e. it lists the C++ call stack) then:

- If the **apl.lines** file is absent or older than the interpreter, the stack trace will show function names and code offsets in hex. For short functions that is sort of OK, but for longer ones the hex numbers can make it difficult to find the exact location of the fault that has triggered the stack trace.
- If the **apl.lines** file is up-to-date (i.e. **make apl.lines** was run after the interpreter binary was produced), then the stack traces will show source file line numbers instead of hex numbers. Producing **apl.lines** takes quite a while, therefore it is not built automatically.

make gen / gen2

These make targets are only available in sub-directory **tools**. Each produces the file **src/Prefix.def**. After every new token it reads, the interpreter has to decide if (and which) a new reducible phrase is present. This check is executed in the inner loop of the interpreter, therefore the efficiency of this check is crucial for the performance of the interpreter.

- GNU APL uses **tools/phrase_gen** to produce **src/Prefix.def**.

- **src/Prefix.def** is therefore automatically generated and **#included** in **Prefix.hh** and in **Prefix.cc** (= the interpreter source code that performs the phrase matching).
- The **phrase_gen** tool can produce two types of code:
 - either a collision-free hash table (the default),
 - or else a search tree.
- After a **make gen** the next compiler build will use the hash table, while after ***make gen2** it will use a search tree. Performance tests of both alternatives have shown only a marginal performance difference between both methods, but the technique as such might be of interest.

Source Code Documentation

GNU APL is fully documented with **Doxygen**. The term **fully** means that every class member has a (typically rather brief) Doxygen description and **not** that the description is exhaustive. The purpose is primarily to be a memory aid for browsing the source code.

The entire **Doxygen** documentation of GNU APL can, provided that **Doxygen** itself is installed, be produced like this:

```
make DOXY
```

The result is around 6,000 files in directory **html**. After generating them, the files can be browsed with e.g.

```
firefox html/index.html
```

or with some other browser. The **Doxygen** documentation is a good starting point for those that want to get a quick overview of the GNU APL source code.

Source Code File Types

The **Makefile*** files belong to the build system and were explained above. What remains are:

- C source files (file extension **.c**) are almost never used,
- C header files (file extension **.h**) are almost never used, a noteworthy exception is **config.h** which is produced by the **./configure** script.
- C++ source files for inline functions (file extension **.icc**),
- C++ source files (file extension **.cc**),
- C++ header files (file extension **.hh**), and
- pure macro files (file extension **.def**).

By and large, the rules for C and C++ source and header files are:

- Small and many rather than large and few (aka. modularity). Early versions of GNU APL did not follow this rule and some left-overs still do not (but those will be further modularized in the future).
- One class per file; the file name reflects the class name. Exceptions are
- classes that are closely related (e.g. classes **Token** and **Token_string**)
- The .cc and .hh file names are the same, except for the file name extension and classes with a small (say, < 100 lines) source footprint (e.g. those in PrimitiveFunction.hh and PrimitiveFunction.cc).
- C++ declarations can be made in the .cc file if their scope is local to the .cc file (like static declarations in C), otherwise they go into the corresponding .hh file.

.def Files

C/C++ source files primarily contain C/C++ code and occasionally declarations. C/C++ header files primarily contain C/C++ declarations and often (inline) C/C++ code. In contrast, .def files contain neither C/C++ code nor C/C++ declarations, They only contain macro calls whereby the definition of the macro is defined outside the .def file. Currently there are about 20 .def files in the GNU APL source code so that they deserve an explanation.

Why .def Files?

Suppose we have a number of integer error codes:

```
enum ErrorCode
{
    NO_ERROR      = 0,
    SYNTAX_ERROR  = 1,
    RANK_ERROR     = 2,
    LENGTH_ERROR  = 4,
};
```

Sooner or later we need a string that describes the error in a human readable format. This requires a mapping between integer error codes and string literals. There are 2 standard methods in C/C++ to define such a mapping.

Method 1: struct definition

```
struct _error_map
{
    ErrorCode ecode;
    const char * ename;
} error_map =
{
    { NO_ERROR,      "OK" },
    { SYNTAX_ERROR,  "Syntax Error" },
    { RANK_ERROR,    "Rank Error" },
    { LENGTH_ERROR,  "Length Error" },
} error_map;

const char *
get_error_string(ErrorCode ec)
```

```
{
    return error_map[ec];
}
```

Method 2: switch statement

```
const char *
get_error_string(ErrorCode ec)
{
    switch(ec)
    {
        case NO_ERROR:      return "OK";
        case SYNTAX_ERROR:  return "Syntax Error";
        case RANK_ERROR:    return "Rank Error";
        case LENGTH_ERROR:  return "Length Error";
    }
    return "Unknown error";
}
```

Now suppose we add another ErrorCode **DOMAIN_ERROR** to the **enum ErrorCode**. Suppose further that we add the new error code to the enum but not to the **get_error_string()** function. This is a fairly common mistake, in particular because **ErrorCode** is usually declared in a **.hh** file and **get_error_string()** is defined in the corresponding **.cc** file. In that case Method 1 will most likely produce a segmentation fault at runtime, which is about the worst case that may happen. <Method 2 is a little better because a smart compiler like g++ will issue a warning in this case. Both methods are, however, quite inconvenient for the C/C++ programmer, in particular if many such mappings are needed in a larger program.

In GNU APL this problem is solved in the following way:

- **#define** a macro that expresses the relationship (between integer error codes and C/C++ literals for the corresponding error text in our example), and
- call that macro, typically in several different files with the macro defined differently. Calling the macro is performed by **#including** the **.def** file. In our example:

In e.g. **errors.def**:

```
error_def(
error_def(NO_ERROR      , 0, OK
error_def(SYNTAX_ERROR , 1, Syntax Error
error_def(RANK_ERROR:   2, Rank Error
error_def(LENGTH_ERROR: 3, Length Error )
```

```
#undef error_def /* to detect missing definitions of the error_def() macro */
```

In, say, **errors.hh**:

```
enum ErrorCode
{
#define error_def(enum_name, enum_value, _error_string) \
    enum_name = enum_value,
```

```
#include "errors.def"
};
```

And finally in **errors.cc**:

```
const char *
get_error_string(ErrorCode ec)
{
    switch(ec)
    {
#define error_def(enum_name, _enum_value, error_string) \
        case enum_name: return #error_string;
#include "errors.def"
    }
    return "Unknown error";
}
```

Note the following:

- the macro invocations of **error_def()** (in the **.def** file) have more arguments than are used in their macro definitions (in the **.hh** or **.cc** files). GNU APL has a naming convention that unused macro arguments of macros (which are typical for the **.cc** and **.hh** files) start with underscore (_). For this reason the argument lists of the **#defines** above differ slightly.
- **error_string** is not quoted in **errors.def** even though **get_error_string()** returns a `const char *`. The quoting is performed by **#** in **#error_string** (a preprocessor feature).
- the example above uses a simple 1:1 relation between two types. Some **.def** files use more than two sides, e.g. numeric IDs, string names, and help texts.

Rules for .def Files

- The **.def** file should **#undef** all macros that it uses. This is a safeguard against forgetting to **#define** a macro before **#including** the **.def** file (this may or may not be detectable by the compiler).
- A **.def** file should normally not use more than one macro. An exception occurs when macros are very similar (e.g. `SystemVariable.def` with macro `ro_sv_def*` (for read-only system variables), **rw_sv_def** (for read/write system variables), and **sf_def** for system functions).
- item separators such as `,` in enums or `;` in case statements are contained in the macro definitions (i.e. in the **.hh** or **.cc** file) and not in the macro call (i.e. not in the **.def** file). This is because the separators may differ for different macros.

The most important take-aways here are:

- **.def** files contain only macro invocations,
- **.def** files are **#included** in several files and almost certainly the macro definitions (before **#including** a **.def** file) differ between the files,
- the macros are **#defined** outside the **.def** file but **#undef'ed** inside the **.def**

file. (This is normally a bad programming practice, but on purpose here).

Coding Style

GNU APL is a GNU project. For the most part it therefore follows the GNU coding standard, see: <https://www.gnu.org/prep/standards/standards.html> With some minor deviations which attempt to further improve the readability of the C/C++ code.

General

- keep lines short (≤ 78 characters) where possible
- NEVER use tabs (ASCII 0x09) in the source code

Indentation and Blanks

Function bodies (.cc files)

- Variable declarations at top-level scope are **not** indented:
- C++ code at the top-level scope is indented with 3 blanks:

```
void
foo()
{
int x = 0;           // declaration: not indented!

    printf("%d:", x); // code: indented by 3 spaces
int y = 1;           // another declaration.
    ...
}
```

- Variable declarations below top-level scope are indented with 2 (from the { starting the scope
- C++ code below the top-level scope are also indented with 2 blanks:

```
void
foo()
{
int x = 0;           // declaration: not indented!
int y = 1;           // another declaration.

    if (x)
    {
        printf("positive x=%d", x); // code: indented by 2 spaces
    }
    else
    {
        printf("zero x");           // code: indented by 2 spaces
    }
}
```

Conditionals

- conditionals are not functions! put 1 blank between the keyword and the condition
- the body of the conditional is indented according to the opening (of the condition (for short ones like if, else, or for), but no more than 2 blanks:

```
void
foo()
{
  int x = 0;

  // DO NOT:
  if(i == 0)

  // DO:
  if (i == 0)
  {
    ...
  }

  for (int i = 0; i < N; ++i)
  {
    ...
  }

  while (i == 0)
  {
    ...
  }
}
```

- you MAY put single statements or very short bodies onto the condition line,
- but NOT mix single- and multi-line styles for **if** and **else** clauses, and
- place the { and } of multi-line bodies on the same column, that is:
- you MUST NEVER NEVER NEVER use K&R style:

```
// DO:

if (x >= 0)    return "positive";
else          return "negative";

// DO NOT:

if (x >= 0)    { return "positive"; }    // redundant { }
else          { return "negative"; }    // redundant { }

if (x >= 0)    return "positive";        // single-line style
else          // multi-line style
{
    error = true;
    return "negative";
}

// HEAVEN FORBID (aka. K&R style)
if (x >= 0) {
    return "positive";
} else {
    return "negative";
}
```

Function arguments

- one blank after every comma and between tokens:

```
// DO:
```

```
int
main(int argc, char * argv[])
{
    ...
}
```

```
// DO NOT ANY OF THE FOLLOWING...
```

```
int
main (int argc, char * argv[]) // extra space before arguments
{
    ...
}
```

```
int
main( int argc, char * argv[] ) // extra spaces around arguments
{
    ...
}
```

```
int
main(int argc, char *argv[]) // no space after *
{
    ...
}
```

```
int
main(int argc, char* argv[]) // no space before *
{
    ...
}
```

Naming Conventions

C++ Class Names

Class names should be in **CamelCase**. That is:

- No `_` between multiple words like in **CamelCase**,
- First letter of each word shall be uppercase, the rest lowercase,
- Abbreviations shall be all uppercase (in that case `_` is allowed if it improves readability or groups classes). E.g. `Quad_XML`.

C++ File Names

- class name and file names shall be the same (except for the `.cc` and `.hh` extensions).
- file names: always use `.cc` (not `.cxx` or `.cpp`) and `.hh` (not `.hxx` or `hpp`).
- Pure C (as opposed to C++) files should use `.c` and `.h` as usual.

Some Coding Conventions

- Use **(const) char *** for ASCII characters (i.e. 0x00...0x7F) and **(const) UTF *** for UTF8 encoded strings. Keep in mind that C++ literals with APL characters (e.g. "AΔB") are automatically UTF8 encoded by the compiler even though their type is **const char ***.
- Use **basic_string<typename>** for simple C types (char, int, char *, ...) and **vector<typename>** for structs and classes, in particular for those with constructors.
- Use **const** where possible.

These conventions have developed over time. Some old code does not follow them, but new or updated code should.

Class declarations (.hh files)

To the extent possible:

- declare public members (= the external interface) before protected members (= the class implementation),
- declare members before static members,
- declare member functions before data members,
- keep the following declaration order:
 - constructors,
 - destructor,
 - inline functions,
 - non-inline functions
- always use **protected:** rather than **private:** for the members of a class. GNU APL uses the convention that **private:** members are used to prevent the accidental function calls that are sometimes generated by the C++ compiler (implicit conversion). In GNU APL, such members are declared, but not defined (so that the linker will complain if the compiler should generate them).

Indentation:

- indent all members with 2 spaces,
- indent inline function bodies by 3 more spaces
- do this recursively (nested classes)

And finally: have a look at the existing **.cc** and **.hh** files. Even though this will cause some extra work on your part, it simplifies the reading of your code for many others. A project with different styles in different files looks rather unprofessional, so please respect the style rules above even if you do not like them.

Debugging

GNU APL has a number of built-in means to locate faults. In a code base of around 120,000 LOC, locating the fault in the source file takes most of the time required to fix it. Many of the GNU APL debugging facilities have performance impacts; they need to be enabled. The make target **develop**, i.e.

```
make develop
```

will enable a useful set of debugging facilities. **make develop** re-runs **./configure**, therefore a normal build must have been performed beforehand (otherwise the top-level Makefile would be missing).

Assertions

GNU APL has more 600 Assertions spread throughout the source code. The assertions check internal assumptions that the designer has made. If such an assumption turns out to be false then a stack dump is produced which tells which assumption (and where) was wrong.

Even though every single assertion has a negligible performance impact, the accumulation of all assertions might become noticeable. It is therefore possible to enable or disable some or all of them. This is done via the **./configure** argument **ASSERT_LEVEL** (see also **README-2-configure**):

```
.configure ASSERT_LEVEL_WANTED=0    (the default, best performance)
.configure ASSERT_LEVEL_WANTED=1    disable trivial assertions, enable others
.configure ASSERT_LEVEL_WANTED=2    enable all assertions (worst performance)
```

The **make develop** target sets, among other things, **ASSERT_LEVEL_WANTED=2**.

Logging Facilities

Failed assertions primarily reveal major programming mistakes, such as changing the code in one place and not considering the impact in other places. In many cases the source code is faulty even though no assertion fails. To isolate a fault one can enable one or more of almost 50 logging facilities from the APL command line. Each logging facility addresses a functional area of interest in the source code. When the code comes across such an area then a printout is produced which provides additional information for the troubleshooter. As of this writing, the following logging facilities exist:

```
]log
1: (OFF) AV details
2: (OFF) new and delete calls
3: (OFF) input from user or testcase file
4: (OFF) parser: parsing
5: (OFF) ...    function find_closing()
6: (OFF) ...    tokenization
7: (OFF) ...    function collect_constants()
```

```

8: (OFF) ... create value()
9: (OFF) defined function: fix()
10: (OFF) ... set_line()
11: (OFF) ... load()
12: (OFF) ... execute()
13: (OFF) ... enter/leave
14: (OFF) State indicator: enter/leave
15: (OFF) ... push/pop
16: (OFF) Symbol: lookup
17: (OFF) ... push/pop and )ERASE
18: (OFF) ... resolve
19: (OFF) Value: glue()
20: (OFF) ... erase_stale()
21: (OFF) APL primitive function format
22: (OFF) character conversions
23: (OFF) APL system function Quad-FX
24: (OFF) commands )LOAD, )SAVE, )IN, and )OUT
25: (OFF) more verbose errors
26: (OFF) details of error throwing
27: (OFF) nabla editor
28: (OFF) execute(): state changes
29: (OFF) PrintBuffer: align() function
30: (OFF) Output: cork() functions
31: (OFF) Details of test execution
32: (OFF) Prefix parser
33: (OFF) ... location information
34: (OFF) FunOper1 and FunOper2 functions
35: (OFF) Shared Variable operations
36: (OFF) command line arguments (argc/argv)
37: (OFF) interpreter start-up messages
38: (OFF) optimization messages
39: (OFF) )LOAD and )SAVE details
40: (OFF) Svar_DB signals
41: (OFF) Parallel (multi-core) execution
42: (OFF) EOC handler functionality
43: (OFF) □DLX details
44: (OFF) command ]DOXY
45: (OFF) details of Value allocation
46: (OFF) □TF details
47: (OFF) □PLOT details

```

Like assertions, debugging facilities may have a performance impact. There are almost 400 places in the source code where a decision is made about if and what debug output shall be produced. For this reason, the logging facilities can either be enabled/disabled at runtime from the APL command line (called **dynamic logging**) or else at compile time (called **static logging**). **dynamic logging** is the method of choice for trouble shooting and is therefore chosen by **make develop**. In contrast **static logging** is the default and the logging facilities that shall be enabled can be set in file **src/Logging.def** (first argument of macro **log_def()**) before compiling the interpreter.

The logging Facility 37, *interpreter start-up messages* is somewhat special. Logging messages produced in the start-up of the interpreter occur before the user has a chance to enter a **ILOG 37** command that enables them. For that reason, the logging of start-up messages can be enabled from the command line (option **-I 37**). None of the other logging facility can be enabled from the command line. In the supposedly rare cases where other facilities are needed before the interpreter has fully initialized itself, static logging can be used instead.

The)CHECK Command

GNU APL has adopted the)CHECK command from IBM APL2. The check command performs an internal check of all data structures inside the interpreter. The primary purpose is to find memory leaks:

```
)CHECK
OK      - no stale functions
OK      - no stale values
OK      - no stale indices
OK      - no duplicate parents
```

A value or function is **stale** if the current workspace cannot reach it anymore. The interpreter does a sort of double-entry accounting of APL values and defined functions, and the)CHECK triggers a consistency check of that accounting. The runtime overhead is low therefore the double-entry accounting cannot be disabled.

Value Tracing

While every logging facility focuses on a particular functional area of interest, value tracing focuses on the life-cycle of individual APL values. If)CHECK finds a stale value (stale functions are rather rare) then the next question is where the value was lost. An APL value sees a lot of places in the source code and the value tracing keeps track of the major steps in the lifetime of a value (creation, copying, destruction, etc). Value tracing is disabled by default (Performance!) but enabled by **make develop**.

Debug macros

If all debug means above fail, then the last resort is to either start a debugger (like **gdb**) or to enter checkpoints in the source code and print the information needed to isolate the fault. For this purpose (the author is using debuggers only for analyzing core dump files) there are some macros that simplify such printouts.

The LOC macro

The macro **LOC** expands to the current source file name and line number. Many C++ functions use it for non-debug purposes to see from where they were called. The expanded LOC macro is a string literal, therefore the overhead is minimal:

```
#define STR(x) #x
#define Loc(f, l) f ":" STR(l)
#define LOC Loc(__FILE__, __LINE__)
```

The Q() macro

The macro **Q(x)** prints the value of **x**, source file name and line number at which the value of **x** was printed:

```
/// print x and its source code location
#define Q(x) get_CERR() << std::left << setw(20) << #x ":" \
```

```
<< " '" << x << "'" at " LOC << endl;
```

The other debug means are somewhat coarse-grained. Dividing around **120,000 LOC** (lines of source code) by 400 places where logging facilities may provide some output yields an average of 300 lines of code between logging outputs. The typical troubleshooting process is this:

1. find an easy way to reproduce the fault (testcase file, APL script, etc.), then
2. use logging facilities and/or value tracing to narrow the possible locations of the fault (to typically one source code file), and finally
3. insert **Q()** macros if needed to further isolate the fault.

The build system checks that all **QO** macros were removed before checking the code into the Savannah SVN repository. This is because forgetting that used to be a frequent mistake.

The Q1() macro

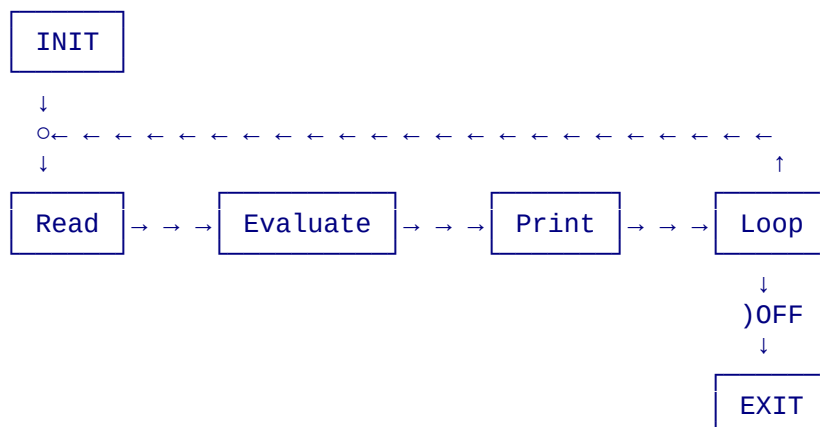
The **Q10** macro does the same as **Q0** but shall remain in the code. It is (rarely) used in places of which the designer believes that they cannot be reached but has no proof that this is the case. Similar to assertions, but only printing values of interest without terminating with a stack dump.

Top-Down Description of the GNU APL Source Code

The APL interpreter is, like almost any other interpreter, a program that reads one line of input after another. For every line read (the **input**) it produces zero or more lines of **output**. In doing so, the interpreter may or may not change its internal state.

In the context of APL the internal state of the interpreter is called (the current) **workspace**; it can be **)SAVE**d into a file and later **)LOAD**ed or **)COPY**ed into another interpreter instance.

The top-level loop of the interpreter described above is commonly referred to as **REPL**, an abbreviation for **Read-Evaluate-Print-Loop**:



The **REPL** of GNU APL is located at the end of function **main()** in source file

main.cc:

```
for (const bool exit_on_error = IO_Files::exit_on_error();;)
{
    const Token tok = Workspace::immediate_execution(exit_on_error);
    if (tok.get_tag() == TOK_OFF)    Command::cmd_OFF(0);
}
```

The **Read**, **Evaluate**, and **Print** parts of the **REPL** are performed in function **immediate_execution()** which returns **TOK_OFF** if the APL command **)OFF** was processed and something else otherwise.

The Workspace::immediate_execution() Function

Function **Workspace::immediate_execution()** processes one line of input, in source file **Workspace.cc**. It simply calls **Command::process_line()**. More importantly, it **catches** C++ errors that may be thrown and processes the errors:

```
Token
Workspace::immediate_execution(bool exit_on_error)
{
    for (;;)
    {
        try
        {
            Command::process_line();
        }
        catch (Error & err)
        {
            ...
        }
    }
}
```



Some lines, notably the ∇ -editor and, in certain cases, the **)COPY** and **)LOAD** commands have their own sub-REPL. They loop themselves until a particular condition (the closing ∇ for the ∇ -editor or end-of-file for the **)COPY** and **)LOAD** commands) has occurred. Therefore:

- process_line() is **atomic** (in the sense that it processes only one input line), but
- the functions called by process_line() will be called repeatedly until an error is returned. The close of the ∇ -editor is also considered an error condition here).

This fact is sometimes missed by programmers using **libapl**. **libapl** is a library that contains most of the normal interpreter code, **but not** the REPL loops of the interpreter. The purpose of **libapl** is to provide a C-API to other front-ends like **Erlang**, **Python**, or **C++** programs. For example, if a front end opens a defined function with a line that starts with ∇ , then the front-end is responsible for providing the subsequent (defined function-) lines including the final ∇ that closes the function definition. The same holds for **)LOAD** and **)COPY** of **.apl** scripts, but not for **)LOAD** and **)COPY** of workspace snapshots (**.xml** files). To summarize:

Operation Atomicity

∇ -editor	non-atomic
------------------	------------

Operation Atomicity

)SAVE	atomic
)LOAD .apl	non-atomic
)LOAD .xm	atomic
l	
)COPY .apl	non-atomic
)COPY .xml	atomic
multi-line string	non-atomic

The Command::process_line() Functions

Function **Command::process_line()** reads the next input line (**InputMux::get_line()**), removes leading white-space (blanks) and then decides how to proceed. The first non-blank character determines the subsequent processing as follows:

1. empty line: return
2. ♂: APL comment: return
3. #: script comment: return
4.): (regular) APL command: **Command::process_line(line...)**
5.]: (debug) APL command: **Command::process_line(line...)**
6. ""|«««: start of a multi-line string: loop until end of string
7. otherwise: APL expression: **Command::process_line(line)**

Function **Command::process_line()** is overloaded; the variant without arguments fetches a line while the variant with a line argument processes the fetched line. The decision about the further processing is done in the latter, i.e. in **process_line(line...I)**:

```
void
Command::process_line(UCS_string & line, ostream * out)
{
    line.remove_leading_white_spaces();
    if (line.size() == 0) return; // empty input line

    switch(line[0])
    {
        case UNI_R_PARENT: // regular command, e.g. )SI
            if (out == 0) out = &COUT;
            do_APL_command(*out, line);
            if (line.size()) break;
            return;

        case UNI_R_BRACK: // debug command, e.g. ]LOG
            if (out == 0) out = &CERR;
            do_APL_command(*out, line);
            if (line.size()) break;
```

```

        return;

    case UNI_NABLA:                // e.g. ∇FUN
        Nabla::edit_function(line);
        return;

    case UNI_NUMBER_SIGN:          // e.g. # comment
    case UNI_COMMENT:              // e.g. ⌘ comment
        return;

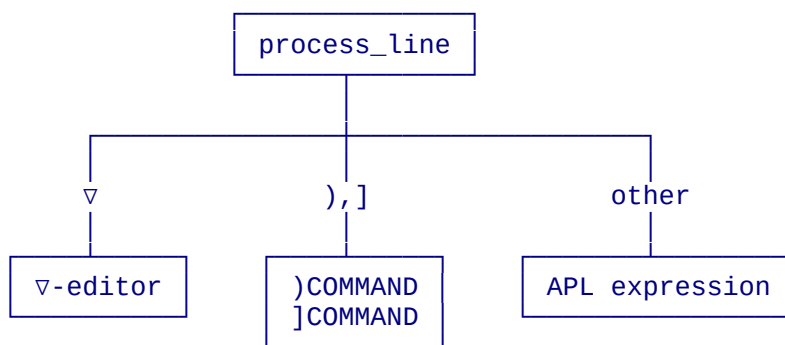
    default: break;
}

++APL_expression_count;
do_APL_expression(line);
}

```

By default the output of `process_line()` is printed to the console, but a non-zero **out** redirects the output to **out**. This is used by functions that need to capture the output of a command, e.g. `⌘"COMMAND"` (a GNU APL extension, normally APL commands cannot be used by `⌘`).

As explained, further processing depends on the first character of the line:



Strictly speaking the `∇-editor` and APL commands are not part of the APL language (they have their own syntax). At this point it should suffice that the implementation of the `∇-editor` is found in source file **Nabla.cc** (nabla is the name for `∇` in the Greek alphabet) while the implementations of most commands are found in **Command.cc**. Most of the functions in **Command.cc** are concerned with the decoding of command arguments and are not further discussed here. Most commands have a simple implementation which is also contained in **Command.cc**. The others have their own source file:

Command Implementation

<code>)CLEAR</code>	<code>Workspace.cc</code>
<code>)MORE</code>	<code>Workspace.cc</code>
<code>)SAVE .xml</code>	<code>Archive.cc</code>
<code>)COPY .xml</code>	<code>Archive.cc</code>
<code>)LOAD .xml</code>	<code>Archive.cc</code>

Consult source file **Command.def** to find the implementation of a particular command. Most commands support TAB-completion: the user hits TAB and the possible completions are displayed. Implemented in **TabExpansion.cc**.

APL Expressions

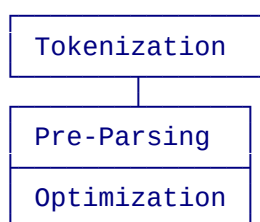
The most interesting case occurs if the user enters an APL expression, from simple ones like **+/10** or complex ones like calling a defined function. The behavior in this case is specified in the ISO APL standard and GNU APL follows this standard, at least for the most part. The processing of the input line is *essentially* the same for the lines of defined APL functions and for lines entered in immediate execution. The minor differences, e.g. APL labels are not allowed in immediate execution, statement separators are not allowed in \downarrow -strings, etc. are handled via a **ParseMode** declared and defined in **APL_types.hh**:

```
/// What is being parsed (defined function, immediate execution statements,
/// or  $\downarrow$ expr)
enum ParseMode
{
    PM_FUNCTION      = 0,    ///< defined function
    PM_STATEMENT_LIST = 1,    ///< immediate execution
    PM_EXECUTE       = 2,    ///< execute ( $\downarrow$ )
};
```

The **ParseMode** defines the origin of a particular line; the options are:

- The line was entered in immediate execution (as described above), or
- The line is a line of a defined function (see later), or
- The line was an APL string (\downarrow , \square EA, \square ES ...).

The next steps in the processing of a line are tokenization and pre-parsing:



Tokenization

Tokenization splits the (APL-) characters of a line into small fixed-sized units called *Token*s. Tokens are the machine-readable unit equivalents of the human readable input characters. For example, a possibly lengthy string like "3.14159" is converted into a Token holding a C *double*. The whole purpose of tokenization is to convert the APL input into an internal representation that can be interpreted significantly faster than the APL characters entered by the APL programmer. For example:

A-1 2 3 \diamond B-'abc'

\uparrow APL input string

is tokenized into:



The function that performs the tokenization is:

```
ErrorCode tokenize(const UCS_string & input, Token_string & tos) const;
```

which is a member of **class Tokenizer**, see **Tokenizer.hh**. The result is a `Token_string`:

```
// A sequence of Tokens
class Token_string : public std::vector<Token>
{
    ...
};
```

see file **Token.hh**.

Pre-parsing

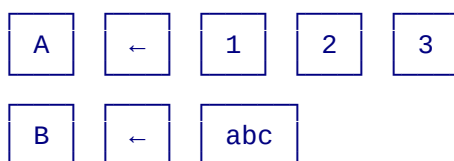
Pre-parsing is primarily an optimization step that performs a number of sub-steps, such as:

- splitting multiple statements (separated by ◇ Token) into individual statements (at ◇),
- grouping multiple scalars into single APL values,
- reversing the order of individual statements, and
- performing some optimizations

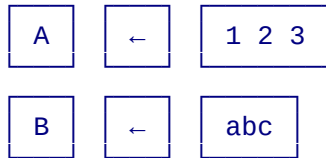
In the example above, the token string



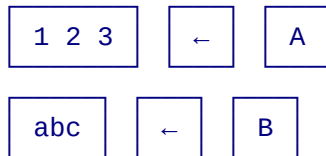
is first split into 2 statements:



The scalars 1, 2, and 3 are then combined into a single APL value (the APL value *abc* is already a single value after tokenization):



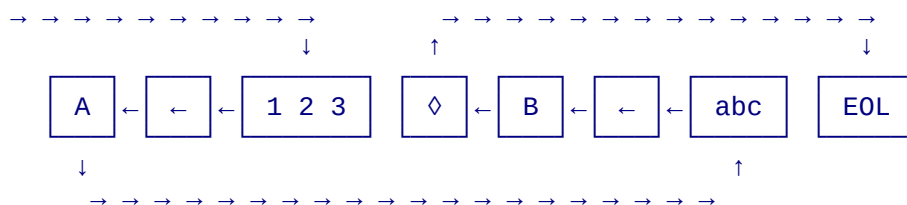
Next, the individual statements are reversed:



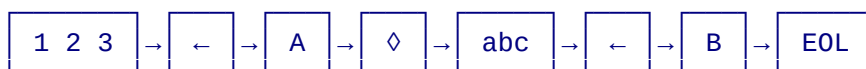
and finally combined again:



The reason for reversing the tokens of every statement is somewhat subtle: APL statements are executed from left to right, but the tokens inside a statement are processed from right to left. Without the reversal the tokens would need to be processed like this:



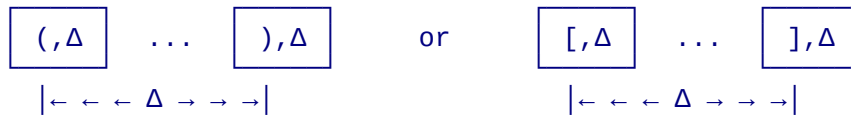
By reversing every statement the same processing order now becomes:



DEFINITION: in source code comments, we refer to the initial token order (statement tokens from left to right, statements from left to right, and lines from top to bottom) as the **APL order** and to the order after reversing the tokens of every statement (statement tokens from right to left, statements from left to right, and lines from top to bottom) as the **reversed order**. The reversal is performed in **Executable::parse_body_line()**. In other words, function ***Tokenizer::tokenize()** produces a **Token_string tos** in APL order, which is later reversed into the final function body **Token_string Executable::body**.

The reversed order can be processed much faster since the order is now linear and no more back and forth jumps are needed when the token string is interpreted.

Another trick in the pre-parsing phase is to store the distance between opening and closing parentheses and square brackets (curly brackets are removed entirely in this phase) in the token itself. Sometimes the prefix-parser (**Prefix.cc**) that finally performs the interpretation has to jump over a pair of parentheses or brackets to find the token to their left. Instead of scanning the token string at runtime until a matching parenthesis or bracket is found, one can compute its location directly (i.e. in time $O(1)$ rather than $O(\Delta)$ for distance Δ between the matching parentheses or brackets):



The Pre-parser function returns an error code and fills its argument **tos** as a side effect:

Parser.hh:

```
/// Parse token string \b input into token string \b tos.
ErrorCode parse(const Token_string & input, Token_string & tos) const;
```

The entire conversion (i.e. tokenization and pre-parsing happens in function **Executable::parse_body_line()**; the result is stored in member **body** of class Executable:

Executable.hh:

```
/// The tokens to be executed. They are organized line by line and
/// statement by statement, but the tokens within a statement are
/// reversed due to the right-to-left execution of APL.
Token_string body;
```

Every defined function, every line entered in immediate execution, and every statement from a Φ expression is converted into an object of base class **Executable**. That object contains some more information such as the original line(s) from which it was created, a reference counter to delete it if no longer used, etc.

From base class Executable three different classes for the different ParseModes are derived:

```
class ExecuteList : public Executable { ... }           // in Executable.hh
class StatementList : public Executable { ... }         // in Executable.hh
class UserFunction : public Function, public Executable // in UserFunction.hh
```

This is because the minor differences between the ParseModes not only affect the parsing of the input line but also some runtime aspects. The base class Executable contains the common functionality while the derived classes contain the specific functionalities related to the different ParseModes.

Some debug printouts display the ParseMode, like this:

- ∇ for defined functions,
- \diamond for a list of statements (i.e. immediate execution), and
- \perp for executed APL strings (\perp , \square EA, \square EB, \square ES).

The difference between \diamond and \perp is mostly resolved during pre-parsing, therefore classes **StatementList** and **ExecuteList** are almost the same as class **Executable**. Both pertain to a single line of APL code and their implementation can be found in **Executable.cc**.

In contrast, defined functions are far more complex (multiple lines, labels, parameter passing) and have their own implementation file **UserFunction.cc**.

Naming of APL objects

To conclude the discussion of pre-parsing we should mention the role of names. Ignoring comments and white-space (which are discarded rather early), the lines of every APL program consists of 3 types of objects:

1. literals (constants) like 1 2 3 or *abc*, and
2. names of built-in primitives like + and -, system functions like \square UCS, and system variables like \square IO
3. (user-defined) names of variables and defined functions, in the following called **symbols**.

These objects are processed by the pre-parser according to the following rules:

1. literals are converted to APL values,
2. every primitive, system function, or system variable has an integer ID which uniquely identifies it, and
3. every user defined name is entered into a symbol table, which is a hash table that contains all user defined names. The conversion from a name (i.e. string) to a user defined object happens during the pre-parsing (and therefore has no runtime overhead).

The IDs of all primitives are **#defined** in **Id.def**. The IDs are sorted by the first character in their name (the character after \square for system functions and variables) or e.g. "P" for Plus, "M" for Minus etc. These IDs are not only used in the body tokens of executables but also in)SAVED .xml snapshots. It is therefore essential that new IDs are added at the end of the ID group that starts with the same character, and that obsolete IDs are kept (or at least their IDs are never reused). The first character (upper 8 bits of the ID are chosen in a way that simplifies debugging at little or no cost.

All user defined symbols have the ID 0x0000 and their token contains a unique **Symbol pointer**. **The symbol table (class *SymbolTable in file SymbolTable.hh) is only used by the pre-parser (to ensure that all symbols have the same Symbol) and by some APL commands like *)FNS,)VARS,)OPS, etc.**

The Id.def file classifies the around 200 IDs via 5 macros that indicate their purpose or usage:

- macro **pp(...)**: **p**retty **p**rint. For constructing the name (usually "---") of a token that has no permanent role, e.g. pass C function arguments. E.g. NO_VALUE, VARIABLE, ...
- macro **qf(...)**: **q**uad **f**unction. Quad function, e.g. □DL, □FX, ...
- macro **qv(...)**: **q**uad **v**ariable. Quad variable, e.g. □CT, □IO, ...
- macro **sf(...)**: **s**ystem **f**unction. An APL primitive, e.g. ρ, Φ,
- macro **st(...)**: **s**ystem **t**oken. An ID for other token, e.g. α, →, ...

Token Structure

With the prerequisites above we can define the structure of a **Token**:

```
Token.hh:

class Token
{
    ...

    /// the (optional) value of the token.
    union sval
    {
        Unicode      char_val;          ///< the Unicode for TV_CHAR
        APL_Integer   int_vals[2];      ///< the integer for TV_INT
        APL_Float_Base float_vals[2];   ///< the doubles for TV_FLT and TV_CPX
        Symbol        * sym_ptr;        ///< the symbol for TV_SYM
        Function_Line fun_line;         ///< the function line for TV_LIN
        IndexExpr      * index_val;     ///< the index for TV_INDEX
        Function_P      function;       ///< the function for TV_FUN
        Value_P_Base    apl_val;        ///< the APL value for TV_VAL

        /// a shortcut for accessing apl_val
        Value_P & _apl_val() const
        { return reinterpret_cast<Value_P &>
            (const_cast<Value_P_Base &>(apl_val)); }

    };

    ...

protected:

    /// The tag indicating the type of \b this token
    TokenTag tag;

    /// The value of \b this token
    sval value;

    ...
}
```

In short this means that:

- a token is a tagged union (named **sval**), and
- the tag (an **enum TokenTag**) determines which member (if any) of **union sval** a particular token holds.

The APL interpreter can be understood as virtual machine whose opcodes are Tokens. In this context the attribute **virtual** of *virtual machine* must not be

confused with the attribute **virtual** of *virtual C++ functions* or *classes*. The tokenization and pre-parsing is then the "compilation" of APL source code into machine code of this virtual APL machine. The rest of **Token.hh** defines quite a number of constructors and access functions of class Token, most of them *inline* because the manipulation of Token is usually simple but performed frequently.

The **TokenTag** is a 32-bit (4 byte) number which has 3 sub-fields:



The ID of a Token

The 16 bit **ID** is the integer ID of system functions and variables explained above. The ID (i.e. its access function Token::get_Id()) is not used during the normal interpretation of APL code but only in error messages and for loading and saving of workspaces.

The Value-Type of a token

The value type simply defines which of the members of **union sval** the token holds. The value type is only used when a Token is copied or moved around (most likely from the body of a defined function into the prefix parser as will be explained below). The knowledge of the type is needed because some the **union sval** members have constructors.

The value types are defined in **TokenEnums.hh**:

TokenEnums.hh:

/// The value type of a token

```
enum TokenValueType
{
    // token value types. The token value type defines the type of the
    // token in the union 'value'.
    //
    //
    TV_MASK          = 0xFF00,
    TV_NONE          = 0x0000,    // value not used
    TV_CHAR          = 0x0100,    // Unicode          .char_val
    TV_INT           = 0x0200,    // uint64_t         .int_val;
    TV_FLT           = 0x0300,    // APL_Float        .flt_val;
    TV_CPX           = 0x0400,    // cdouble          .complex_val;
    TV_SYM           = 0x0500,    // Symbol *         .sym_ptr;
    TV_LIN           = 0x0600,    // Function_Line    .fun_line;
    TV_VAL           = 0x0700,    // Value_P          .apl_val;
    TV_INDEX         = 0x0800,    // IndexExpr *     .index_val;
    TV_FUN           = 0x0900,    // Function_P       .function;
};
```

The typical (inline) function that copies a Token is:

Token.hh:

```
Token::copy_N(const Token & src)
```

```

{
    tag = src.tag;
    switch(src.get_ValueType())
    {
        case TV_NONE:  value.int_vals[0]    = 0;
                       value.int_vals[1]    = 0;                                break;
        case TV_CHAR:  value.char_val       = src.value.char_val;                break;
        case TV_INT:   value.int_vals[0]    = src.value.int_vals[0];
                       value.int_vals[1]    = src.value.int_vals[1];                break;
        case TV_FLT:   value.float_vals[0]  = src.value.float_vals[0];            break;
        case TV_CPX:   value.float_vals[0]  = src.value.float_vals[0];
                       value.float_vals[1]  = src.value.float_vals[1];            break;
        case TV_SYM:   value.sym_ptr        = src.value.sym_ptr;                  break;
        case TV_LIN:   value.fun_line       = src.value.fun_line;                  break;
        case TV_VAL:   value._apl_val()     = src.value._apl_val();                break;
        case TV_INDEX: value.index_val      = src.value.index_val;                 break;
        case TV_FUN:   value.function       = src.value.function;                 break;
        default:       FIXME;
    }
}

```

In **copy_N()**, **src** is a token that is copied into token **this**, and **value** is the member (of **union sval**) that is being copied.

The TokenClass-Type of a token

The most important part of a **TokenTag** is its class (not to be confused with C++ classes). Like the token value types they are defined in **TokenEnums.hh**. There are two different kinds of token classes: **permanent token classes** and **temporary token classes**. The permanent classes are:

TokenEnums.hh:

```

enum TokenClass
{
    // token classes.
    //

    // permanent token classes. Only Token of these classes can appear in
    // the body of a defined function.
    //
    TC_ASSIGN          = 0x01,    ///< ←
    TC_R_ARROW         = 0x02,    ///< →N
    TC_L_BRACK         = 0x03,    ///< [ or ;
    TC_R_BRACK         = 0x04,    ///< ]
    TC_END             = 0x05,    ///< left end of statement
    TC_FUN0            = 0x06,    ///< niladic function
    TC_FUN12           = 0x07,    ///< ambivalent function
    TC_INDEX           = 0x08,    ///< [...]
    TC_OPER1           = 0x09,    ///< monadic operator
    TC_OPER2           = 0x0A,    ///< dyadic operator
    TC_L_PARENT        = 0x0B,    ///< (
    TC_R_PARENT        = 0x0C,    ///< )
    TC_RETURN          = 0x0D,    ///< return from defined function
    TC_SYMBOL          = 0x0E,    ///< user defined name
    TC_VALUE           = 0x0F,    ///< APL value

    TC_MAX_PERM,                ///< permanent token are < TC_MAX_PERM
    ...
}

```

In contrast, temporary tokens are never used in the body of an **Executable**. They are short-lived and vanish after some permanent token was created from them. For

example, prefix parser creates a token of class **TC_PINDEX** (for partial index) when it sees the closing **]** of an APL index (remember: right-to-left) and creates a permanent token of class TC_INDEX when it sees the corresponding opening **[**. Likewise, a defined function without result returns a token of class TC_VOID which is passed from one prefix parser to another to indicate the result of a defined function.

In short, temporary token classes are only created and consumed inside the prefix parser. The temporary token classes are:

TokenEnums.hh:

```
...
// temporary Token classes. Token of these classes only appear as
// intermediate results during tokenization and prefix parsing
//
TC_PINDEX      = 0x10,    ///< partial index
TC_VOID        = 0x11,
TC_MAX_PHRASE,          ///< token in phrases are < TC_MAX_PHRASE
TC_MAX_PHRASE_2 = TC_MAX_PHRASE*TC_MAX_PHRASE,    // TC_MAX_PHRASE ^ 2
TC_MAX_PHRASE_3 = TC_MAX_PHRASE*TC_MAX_PHRASE_2,  // TC_MAX_PHRASE ^ 3
TC_MAX_PHRASE_4 = TC_MAX_PHRASE*TC_MAX_PHRASE_3,  // TC_MAX_PHRASE ^ 4

TC_OFF         = 0x12,
TC_SI_LEAVE    = 0x13,
TC_LINE        = 0x14,
TC_DIAMOND     = 0x15,    // ◇
TC_NUMERIC     = 0x16,    // 0-9, -
TC_SPACE       = 0x17,    // space, tab, CR (but not LF)
TC_NEWLINE     = 0x18,    // LF
TC_COLON       = 0x19,    // :
TC_QUOTE       = 0x1A,    // ' or "
TC_L_CURLY     = 0x1B,    // {
TC_R_CURLY     = 0x1C,    // }
```

The phrase table (or phrase tree) described in [The Phrase Table](#) below maps particular sequences of token classes to reduce functions. To make the phrase table a little more readable, we define some shorter names (SN_xxx) for token classes that better reflect the relation between the APL point of view and the C++ representation of it, and/or abbreviates them:

TokenEnums.hh:

```
...
// shorter token class aliases for the phrase table
//
SN_A           = TC_VALUE,
SN_ASS         = TC_ASSIGN,
SN_B           = TC_VALUE,
SN_C           = TC_INDEX,
SN_D           = TC_OPER2,
SN_END         = TC_END,
SN_F           = TC_FUN12,
SN_G           = TC_FUN12,
SN_GOTO        = TC_R_ARROW,
SN_I           = TC_PINDEX,
SN_LBRA        = TC_L_BRACK,
SN_LPAR        = TC_L_PARENT,
SN_M           = TC_OPER1,
SN_N           = TC_FUN0,
SN_RETC        = TC_RETURN,
SN_RBRA        = TC_R_BRACK,
SN_RPAR        = TC_R_PARENT,
```

```

SN_V      = TC_SYMBOL,
SN_VOID   = TC_VOID,
SN_       = TC_INVALID

```

For example: the ISO APL standard defines a phrase table of 39 phrases. In particular phrase 7 is defined as:

7 EVALUATE DYADIC FUNCTION (<A F B> R)

In GNU APL code the ISO phrase **<A F B>** is represented as 3 short-names **SN_A**, **SN_F**, **SN_B**. To further simplify matters, the **SN_** prefix is hidden by means of some macro magic, so that the same ISO phrase in the GNU APL phrase table (in **Prefix.def**) becomes:

```

Prefix.def :

PH( A F B      , A_F_B_      , 0x03CEF , 33 , 0 , 3), // [EA]

```

Given that **SN_A** and **SN_B** are both short names for **TC_VALUE**, and that **SN_F** the short name for **TC_FUN12**, after expanding macro PH() in the C++ preprocessor this translated to something like:

If a sequence of token classes TC_VALUE TC_FUN12 TC_VALUE is discovered, then call C++ function reduce_A_F_B().

Evaluation of APL Expression

The Phrase Table

The IBM APL Language Reference Manual describes the fundamental rule for the evaluation of APL expressions as follows:

All functions execute according to their position within an expression. The rightmost function whose arguments are available is evaluated first.

In parser terms, one could just as well formulate this as:

- execution of an APL expression starts with an empty sequence of token classes,
- the tokens in the sequence are then, from right to left, prepended to the sequence, and
- whenever a prefix of the current sequence is found in the phrase table:
 - the reduce function in the phrase table entry is called with the token of the prefix as arguments,
 - the prefix in the sequence is replaced with the result of the reduce function.

Since GNU APL stores the token in reverse order as explained earlier, from right to left actually means left to right for the token string that is being interpreted. The rules above amount to what is commonly called a LALR(1) parser (short for **Left-Right-LookAhead-Parser** with 1 lookahead token). In GNU APL terminology it

is the **Prefix parser** in file **Prefix.cc**. The somewhat simplified algorithm of the GNU APL prefix parser is:

1. Every Prefix parser starts execution at the beginning of some token string, say **body**, which is in most cases the body of a defined function.
 1. Its integer variable **PC** (actually of type **enum Function_PC**) is initialized to 0 (start of function), and
 2. the sequence of tokens read so far is cleared. Then:
2. The next token is read and PC is incremented.
3. If the token class is **TC_SYMBOL** (i.e. a name defined by the APL programmer) then the new token is replaced by the current value of the symbol, which is one of:
 1. **TC_VOID** - the symbol name was used (tokenized) but not assigned (e.g. the initial values of local variables of a defined function),
 2. **TC_VALUE** - an APL variable,
 3. **TC_FUN0** - a defined niladic function,
 4. **TC_FUN12** - a defined monadic or dyadic function,
 5. **TC_OPER1** - a defined monadic operator (i.e. 1 function argument), or
 6. **TC_OPER2** - a defined dyadic operator (i.e. 2 function argument). This name resolution step happens in function **Symbol::resolve_right()** for symbols right of **←** or in **Symbol::resolve_left()** for symbols left of ***←**. The case **TC_VOID** happens when a name was tokenized, but no value or function was assigned to it.
7. If the new token together with the sequence matches a phrase (as discussed above) then the phrase is reduced (called REDUCE in parser terminology).
4. Otherwise (no phrase matches) the new token is prepended to the sequence and the next token is read (called SHIFT in parser terminology). In this case execution continues with step 2. above.
5. If a phrase with an APL jump, such as **→N** (i.e. to line **N** of a defined function) is matched, then a new value for the **PC** is computed so that **body[PC]** is the first token of line N. If no such line exists, then the Prefix parser returns its result (either TOK_VOID for functions that return no result or TOK_VALUE otherwise).
6. If the parser sees a closing parenthesis **)** or bracket **]** then the parser creates a sub-parser and execution is continued (with the current PC) in the sub-parser until the corresponding opening parenthesis **(** or bracketed **[** is reached. The result of the sub-parser, typically a token with class **TC_VALUE** for parentheses and **TC_INDEX** for brackets is returned to the calling parser,
7. an end of statement token (class TC_END) does not stop the parser. The necessary actions, e.g, printing of the value for non-committed APL values is performed in the reduce function (which calls

StateIndicator::statement_result()), and the parser simply continues after the reduce function returns. Only the end of the *body stops the parser, and the last token in the **body performs the transfer of the value computed by a defined APL function to its caller.**

The tokens in the current sequence are of C++ type **Token_loc** instead of type **Token**. **Token_loc** is a Token and a range in the **body**. When a token is first fetched from, say, **body[PC]** then the range of its **Token_loc** is **[PC:PC+1]**. As more and more consecutive tokens are being reduced, that range grows and contains all tokens that have contributed to the creation of every token. The range in the body of a defined function is needed for proper error reporting.

The case 3. above is the most tricky one and deserves a closer look at it.

Ambiguity of APL symbols.

In compiled languages like C/C++ every name defined by the user must be *declared*. Any use of a name without a prior declaration is an error detected by the compiler. The declaration determines, among other things, whether the name refers to a variable, or to a function, and in that latter case the arity (number of arguments) of a function.

In contrast, in APL the role of a user defined name is not defined at \square FX-time but at runtime. Due to the declarations in e.g. C/C++ the scope of a name (and from that the relevant declaration) can be determined statically (i.e. at compile time) while in APL the role of a name must be inevitably determined at runtime. In both cases (APL vs. C/C++) the same name may have different roles at different times, but the point in time where the role is determined varies. As a consequence, the compiler uses its symbol table only at compile time while APL also uses it at runtime. Consider the following example:

```
)CLEAR

⌹ define function LEAF (for a function that does not call others)
⌹
∇Z←LEAF
  Z←'LEAF is a niladic defined function.'
∇

⌹ define function F00, which localizes symbol LEAF
⌹
∇Z←F00;LEAF
  LEAF←'LEAF is a (localized) variable.'
  BAR
∇

⌹ define function BAR which calls LEAF.
⌹
∇Z←BAR
  Z←LEAF
∇

⌹ then see what happens
⌹
F00 ⋄ BAR
LEAF is a (localized) variable.
LEAF is a niladic defined function.
```

In the simple example above, **BAR** is called twice: first from **FOO** and then directly from immediate execution (to simplify the example; the same would happen if **BAR** were called from some other defined function). The problem in the example above is that **BAR** cannot statically decide whether the symbol **LEAF** in its function body relates to the niladic function **LEAF** or to the local variable **LEAF** of function **FOO**. One can even construct more complex examples where *the same statement* of a defined function must, according to the evaluation rules of APL, produce *different results* depending how it was called. As a

Corollary: The parse tree of an APL statement is only unique when the statement contains no user defined symbols. And therefore:

Corollary: In general, every APL statement yields a (possibly degenerated) parse tree, where each APL symbol in the statement is a branching point at which it has to be decided (at runtime) which branch down the tree shall be chosen.

The degenerated case is the one where all symbols can be resolved statically (such as labels in a defined function), and then the tree becomes a linear sequence. In real-life APL programs this degenerated case is fairly rare. It could be optimized in a simple manner, but its rarity seems not be worth the effort. A more frequent exception is non-conditional branches like **→LABEL** which are being optimized in GNU APL.

Now, back to the **TC_SYMBOL**. If the prefix parser reads a symbol, then it consults the symbol table for the current meaning of the symbol. There are quite a few different cases:

- The symbol is undefined, for example a local variable before a value is assigned to it or a defined function with that name was created. In this case:
 - If the symbol is referenced (i.e. on the right of **→**) then a SYNTAX ERROR is raised, otherwise
 - The symbol is (the name of) a variable that is being assigned (either completely, or indexed, or selectively).
- The symbol is a defined function, in that case the **TC_SYMBOL** token is, depending on the signature of the defined functionⁱ, replaced with a token of class **TC_FUN0**, **TC_FUN12**, **TC_OPER1**, or **TC_OPER2**. The token value then points to a **UserFunction**. Note that APL primitives and APL system functions are not user-defined, and therefore no symbol table lookup is required. The pre-parser can therefore produce the corresponding tokens directly.
- The symbol is a variable.
 - A variable right of **←** is referenced and the **TC_SYMBOL** token is replaced by a **TC_VALUE** token pointing to the current value of the variable. Otherwise the symbol is subject to an assignment and is, for the moment, left unchanged. At a later time one of the assignment phrases is supposed to match:
 - **A←VALUE** [⊖] normal assignment (symbol **A** may or may not exist),
 - **(A B ...)←VALUE** [⊖] vector assignment (symbols **A**, **B**... may or

may not exist),

- **A[X]←VALUE** \Rightarrow indexed assignment (**A** must be a variable),
- **(f... A)←VALUE** \Rightarrow selective assignment (**A** must be a variable),

All this happens in **Prefix::push_Symbol()** which delegates most of its work to either **Symbol::resolve_left()** or **Symbol::resolve_right()**, depending on the position of the symbol in the statement. This position is tracked with **Prefix::set_assign_state()**, which is one of:

APL_types.hh:

```
...
// the state of an assignment
enum Assign_state
{
    ASS_none      = 0,    ///< no assignment (right of ←)
    ASS_arrow_seen = 1,    ///< ← seen but no variable yet
    ASS_var_seen   = 2,    ///< var and ← seen
    ASS_unknown    = 3,    ///< not known (too much effort to determine)
};
...
```



Terminology: The GNU APL prefixes are called **pattern** or **phrase** in the ISO standard. The token sequence above is called **stack** in the ISO standard and also in some of GNU APL debug printouts.

The main implementation of the algorithm above is C++ function

Prefix::reduce_statements(). It returns a **Token**. The function uses C/C++ goto statements because that leads to much clearer code than the so called "structured programming" would. The goto labels in this function are:

- **grow:** do a SHIFT (if no phrase has matched),
- **again:** repeat phrase matching (after a reduce function was called), and
- **done:** the parser returns. The point of **again:** is that the reduction of some phrases, for example an expression in parentheses, may produce a stack that could not be reduced before but now can without reading another token.

An Example

Consider the APL code from our example above, for simplicity in immediate execution:

A←1 2 3

The tokenizer translates the string "A←1 2 3" into a 4-item **Token_string**:



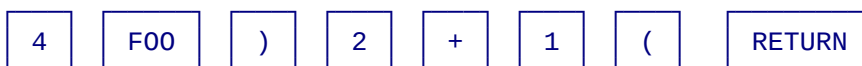
To be precise, the **RETURN** token is actually appended by function **StatementList::fix()** and not by the Tokenizer itself. Then:

1. The prefix parser reads (the single) token **1 2 3** and shifts Pattern is now: **value(1 2 3)**.
2. The prefix parser reads **←** and shifts. Pattern is now: **← value(1 2 3)**.
3. The prefix parser reads **A**. The pattern matches and the parser reduces with **Prefix::reduce_A_ASS_B_()**. The result is **committed_value(1 2 3)** and the new pattern is also: **committed_value(1 2 3)**
4. The prefix parser reads **RETURN**. The pattern matches and the parser reduces with **Prefix::reduce_END_B_()**, which calls ***StateIndicator::statement_result()** and clears the pattern to empty.
5. The parser has no more token to read and returns.

Another Example

(1 + 2) FOO 4 Ⓐ with dyadic defined function FOO

The tokenizer produces:



1. The parser shifts **4** and **FOO** and creates a sub-parser for **)**.
2. The sub-parser reduces **(1+2)** and returns the value **3**.
3. The parser reads the result **3** and computes **3 FOO 4**.

Properties of the reduce_XXX() functions

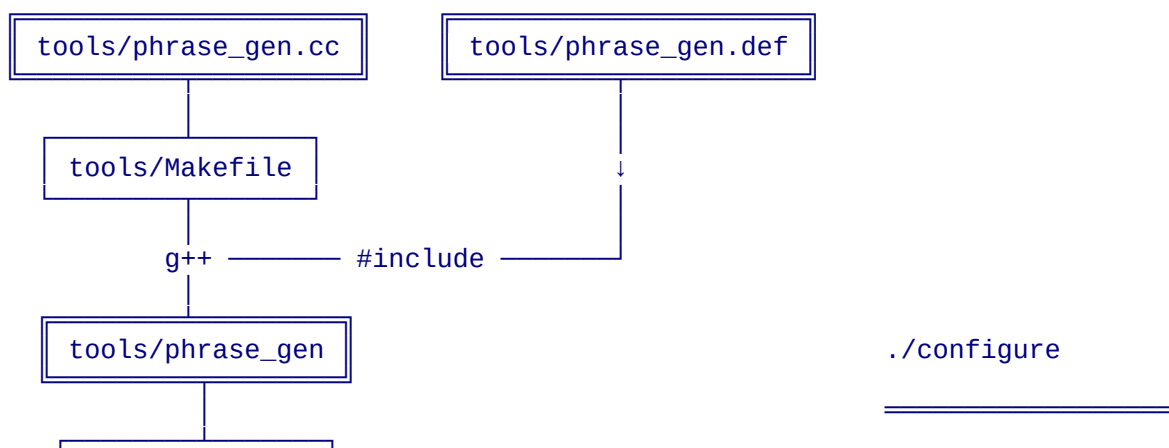
- In GNU APL there is a total of 67 phrases and consequently 67 reduce functions.
- reduce functions are typically short. Nevertheless they cannot be inlined since they are being called via the function pointer **Prefix::Phrase::reduce_fun** in the phrase table (or tree).
- reduce functions have return type **void** and no arguments. They access the token which they reduce directly from a ring-buffer named **Prefix::content** and also store the result there (function **Prefix::pop_args_push_result()**)
- accessing the token is performed via functions **at0()**, **at1()**, **at2()**, or **at3()**, where **at0()** is the leftmost (in APL code order) and the rightmost (in reversed order) token. For example, in **reduce_V_ASS_B()** corresponds **V** to **at0()**, **ASS** to **at1()**, and **B** to **at2()**.
- the lookahead token of the LALR(1) parser is always a token in the body of a defined function. It therefore needs not (yet) to be copied at the time of the lookahead. It is instead accessed with **Prefix::lookahead()**, which returns ***TOK_VOID** if no lookahead token exists.

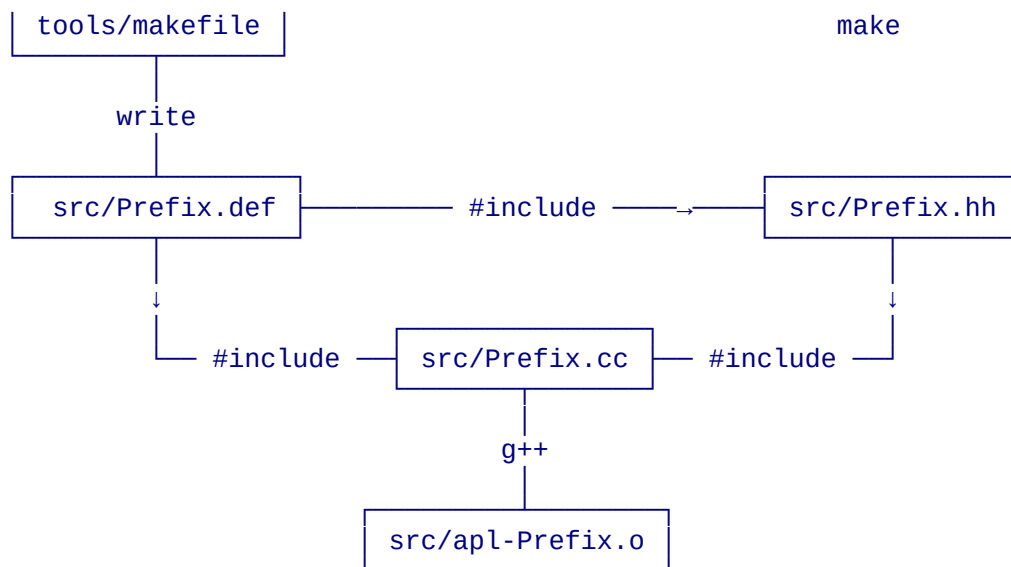
The principal structure of a reduce function is always:

1. obtain the arguments (at0(), at1(), ...)
2. compute the desired side effect (assign a variable, evaluate a function, perform a branch, etc.)
3. remove (pop) the arguments from the stack and push the result on the stack, in most cases with **Prefix::pop_args_push_result()**, and finally
4. set member **Prefix::action** to the next action that the prefix parser shall perform. The possible actions are (enum **R_action** in **Prefix.hh**):
 1. **RA_CONTINUE**: repeat the phrase matching with the current stack (after a **reduce_XXX()** has modified it, e.g. with **pop_args_push_result()**).
 2. **RA_PUSH_NEXT**: push the next token onto the stack (aka. SHIFT)
 3. **RA_RETURN**: return from the current (sub-) parser and push its result onto the stack of the calling parser, or
 4. **RA_SI_PUSHED**: like **RA_RETURN** but without a result (called a **future** in esoteric computer languages). It indicates that a new parser with a new defined function was started and will or will not return a result. This case is slightly different from simply calling a sub-parser, because a sub-parser (e.g. for parentheses) remains in the same function body, while a new defined function has its own body.

Construction of the phrase table (or tree)

The large number of phrases makes it error prone and cumbersome to handle them. For that reason the declaration of all **reduce_XXX()** functions (in **Prefix.hh**, as well as the instantiation of a hash table for fast lookup of phrases (or, alternatively a search tree) in **Prefix.cc** are realized by including **src/Prefix.def**. That is, **Prefix.def** is, with the help of the C++ preprocessor, a code generator for **Prefix.hh** and **Prefix.cc**. **Prefix.def** itself is also generated (with **tools/phrase_gen**). And **tools/phrase_gen** itself **#includes** file **tools/phrase_gen.def** which contains the primary definition of all phrases (similar to the phrase table in the ISO standard. This machinery is hidden behind **tools/makefile** with targets: **gen** (for a hash table) and **gen2** (for a search tree). Note the lowercase **makefile**; the tools directory also contains an (uppercase) **Makefile** which belongs to the build system. The full picture is this:





GNU APL is shipped with **src/Prefix.def** already produced, so that the "normal" designer is rarely concerned with the above. Unless she needs to change the GNU APL phrase table.

If, however, **tools/phrase_gen.def** was modified, then

```

make gen    # generate new src/Prefix.def with a phrase table, or
make gen2   # generate new src/Prefix.def with a phrase tree

```

needs to be performed once. The build system will then note the changed **src/Prefix.def** and a top-level make will recompile the affected files in directory **src**.

Noteworthy Classes

String Classes

The GNU APL source code has more than 1000 places where strings are used. These strings come in 2 flavors:

1. strings of 32-bit Unicodes (sometimes referred to as UCS-32),
2. strings of bytes in UTF8 encoding.

GNU APL uses 2 classes for these strings: class **UCS_string** for 32-bit Unicodes and **UTF8_string** for UTF8-encoded strings. In general:

- UCS_strings are far easier to use than UTF8_strings because UTF8 is a variable-length encoding, which makes indexing or iterating over them cumbersome and error-prone. For this reason all strings used internally in GNU APL are UCS_strings.
- UTF8_strings are commonly used between GNU APL and its environment. They are shorter when transported over the internet, therefore most web pages are UTF8-encoded. Also, filenames in operating systems are UTF8 encoded. Finally, most terminal emulators, in particular those that support APL in some way, send and expect all characters to be UTF8-encoded.

UCS_string

The starting point is:

```
Unicode.hh:

enum Unicode
{
#define char_def(name, uni, _tag, _flags, _av_pos) UNI_ ## name = uni,
#define char_uni(name, uni, _tag, _flags)          UNI_ ## name = uni,
#include "Avec.def"

    Unicode_0      = 0,          ///< End of unicode string
    Invalid_Unicode = 0x55AA55AA, ///< An invalid Unicode.
};
```

where:

- The **char_def()** macro defines all Unicodes that are contained in the 256 character □AV of GNU APL. In old APL1 the □AV and its derivatives (□FC, □A, □D, ...). These days, the □AV is only maintained for backward compatibility with APL1. In APL2 it has been entirely obsoleted by □UCS. Keep in mind, however, that:
 - The encoding of files produced with)OUT or read with)IN are still based on □AV. To be precise: based on some IBM □AV defined in the IBM APL2 reference manual, not on the GNU APL □AV. Generally □AVs as well as their encodings are machine specific, some mainframes even used EBCDIC and older PC operating systems used special APL code pages which corresponded to the □AV.
 - GNU APL has two helpers for such conversions, see below.
 - class *Avec contains numerous static functions for handling and classifying of Unicodes contained in the □AV.
- In contrast, the **char_uni()** macro defines Unicodes that are used in the GNU APL source code but are not contained in the □AV of GNU APL.
- In short:
 - any Unicode with APL significance (primitives etc.) are (and must be) contained in the □AV of the interpreter. These Unicodes are defined with macro **char_def()** in file **Avec.def**, while
 - any Unicode used in the GNU APL C++ source code is defined with macro **char_uni()**.
 - Noteworthy examples of the latter are the sub- and superscripts (⁰, ¹, ², ..., ⁹, and ₀, ₁, ₂, ..., ₉). They are not contained in the □AV of GNU APL but occur in the .xml files produced by GNU APL (and not only in the source code).

The helpers mentioned above are:

```
Avec.hh:
/// a pointer to 256 Unicode characters that are exactly the APL2 character
/// set (□AV) shown in LRM Appendix A, page 470. The □AV of GNU APL is
```

```

    /// similar, but contains characters like ≠ that are not in IBM's EBCDIC
    /// IBM's EBCDIC is used in the )IN command
    static const Unicode * IBM_quad_AV();

    /// recompute \b inverse_ibm_av from \b ibm_av and print it
    static void print_inverse_IBM_quad_AV();

```

After this digression into Unicodes we can now:

```

    UCS_string.hh:

class UCS_string : public std::basic_string<Unicode>
{
    ...
    UCS_string(const UTF8_string & utf);
    ...
}

```

UCS_string has many constructors, most importantly the constructor from a UTF8_string which converts the UTF8_string into a UCS_string. In addition the UCS_string string class defines numerous convenience functions for the processing of UCS_strings.

UTF8_string

In good old C, there were C literals like **"Hello, world"**. At that time the only relevant encoding of such literals was the 7-bit ASCII code and all literals contained only positive (signed) characters.

Next, C++ defined a template class **std::basic_string<>** which provides numerous string processing methods (indexing, iteration) for **basic_string** that are also used in our derived classes **UCS_string** and **UTF8_string**. So far so good.

But now comes the problem. The C++ standard string library says:

```

class string : public basic_string<char> { }

```

On almost all platforms (read: compilers), a **char** is **signed** as opposed to type **unsigned char**. Often this does not hurt because C and C++ automatically convert between the signed and unsigned variants and not every programmer is always aware of the dire consequences of signedness. Like, for example:

```

!
char(-56) = char(200) < char(100) = unsigned char(100) < unsigned char(200).

```

to make things worse, UTF8 encoding works on (unsigned) bytes while C/C++ literals in compilers are, without specific qualifiers, signed. As a consequence, the APL character **←** has Unicode 8592 aka. 0x2190, and a UTF8 encoding of (hex) **E2 86 90** which makes the string literal "←" a (0-terminated) sequence of 3 signed (!) and unfortunately negative (!) characters. Early versions of GNU APL were using **std::string** instead of **UTF8_strings**, and the problems caused by that were numerous. These days GNU APL itself is clean when it comes to signed vs.

unsigned chars, but some older contributions may still suffer. Comparisons for equality works well for mixed signed/unsigned operands, but < and > do not, and, as a consequence, **strcmp()** and friends frequently return the wrong order when used for sorting UTF-encoded strings.

Like **UCS_string** has a constructor from a **UTF8_string**, **UTF8_string** has a constructor from a **UCS_string**:

```
UTF8_string.hh:

/// one byte (not character)! of a UTF8 encoded Unicode (RFC 3629) string
typedef uint8_t UTF8;

class UTF8_string : public std::basic_string<UTF8>
{
    ...
    /// constructor: copy of UCS string. The Unicodes in ucs
    /// will be UTF8-encoded
    UTF8_string(const UCS_string & ucs);
};
```

The take-aways for designers and contributors are:

- prefer UCS_string over UTF8_string where possible (efficiency),
- never use std::string, but
- instead use UTF8_string
- do not write your own conversion functions but use the constructors instead.

APL Values

The primary business of an APL interpreter is to manipulate APL values. An APL value is simply a **Shape** and a **Ravel**, although simplicity ends here.

In that, **Shape** is a class of its own, defined in **Shape.hh**. It has a bunch of constructors for various purposes and also numerous inline convenience functions that makes the code that deals with shapes more readable.

Many of the Shape methods have trivial wrappers in class Value, because almost every shape is accessed in the context of a value. For example:

```
Shape.hh:

...
/// return the rank
uRank get_rank() const
{ return rho_rho; }

Value.hh:

...
/// return the rank of \b this value
uRank get_rank() const
{ return shape.get_rank(); }
...
Shape shape;
...
```

The ravel is simply a C++ array of type **Cell**:

```
Value.hh:
```

```
...  
/// The ravel of \b this value.  
Cell * ravel;
```

```
/// the cells of a short (i.e. p,value ≤ cfg_SHORT_VALUE_LENGTH_WANTED) value  
Cell short_value[cfg_SHORT_VALUE_LENGTH_WANTED];
```

Many real-life APL values are short. To avoid unnecessary memory allocations, GNU APL allocates short ravels inside the class instance and longer ravels outside (which then requires an additional new/delete for the ravel). Put differently, a short value has **ravel == &short_value**, while **ravel** in long values point to a separately allocated memory area. The border between short and long values can be changed with **./configure**; larger values of **SHORT_VALUE_LENGTH_WANTED** have a (marginally) better performance, but the price is a (less marginal) increase in memory consumption.

Similar to **class Shape**, numerous constructors and (mostly trivial) access functions for various purposes. For example, a Value can be constructed from a UCS_string (and conversely a UCS_string can be constructed from a Value (provided that its rank and Cell types are correct). Most values are constructed sequentially and to simplify that, every value has a built-in iterator **Value::next_ravel()** which returns a pointer to the next not-yet-initialized Cell (if any) and 0 after the last Cell of the ravel was returned.



A frequent pitfall related to the **Value::next_ravel()** iterator is to (mistakenly) use it for empty values.

Empty APL values like **0p0** are never empty at the C++ level because every APL value **V** has a *prototype* **↑0pV** whose ravel has one item stored in **ravel[0]**. An old APL joke was, *How many empty arrays does it take to fill an APL workspace? One, if it's big enough.* If a value **V** is empty, then **pV** is 0 (i.e. its ravel has length 0) but length 1 at C++ level (due to its prototype). For this reason, a **Value** has a **Value::element_count()**, which is the ravel length in APL, and a **Value::nz_element_count()** which is the ravel length in C++. The **Value::next_ravel()** iterator uses the former, which causes a segfault when used with an empty value. In short this means that APL values may be empty, but their C++ implementation are never empty *and need to be properly initialized*.



This implies that a proper source code **loop()** over a ravel MUST either use **Value::nz_element_count()** (so that the prototype is initialized inside the loop), or else must have ruled out the empty case beforehand (and may then use **Value::element_count()** safely).

The typical design patterns are:

```
Value_P Z(...);    // may be empty
```

```
    loop(z, Z->nz_element_count())    // not element_count() !!!  
    {  
        new (Z->get_wravel(z)) XxxCell(...);    // safe, but (next_ravel_XXX() is  
not)
```

```
}
```

and:

```
Value_P Z(...);    // may be empty

if (Z->element_count())    // not Z->nz_element_count() !!!
{
    // Z is not empty
    loop(z, Z->element_count())    Z->next_ravel_XXX(...)
}
else
{
    new (Z->get_wproto()) XxxCell(...));    // init prototype
}
```

Z→get_wproto() is the same as **Z→get_wravel(0)** and is used here to make the role of **get_wravel(0)** as the prototype of Z explicit.

class Value_P

APL values are rarely copied but pointers to APL values are passed back and forth a lot in the interpreter (in more than 1200 source code locations), It is therefore easy to loose a Value, which then causes a memory leak. These memory leaks were a real plague in early GNU APL versions, but are now fixed using several counter-measures:

- C++ overloading of operators `::new()` and `::delete()`. Every newly allocated Value, no matter if long or short, is tracked in a link list called **DynamicObject::all_values**, see **DynamicObject.hh**
- The APL command `)CHECK` compares the values that are reachable from APL, e.g. via token of class `TC_Value`, with the values in **DynamicObject::all_values** and reports any mismatches as **stale values**.
- Every value tracks itself with a **ValueHistory** which is, for performance reasons, disabled by default but can be enabled with **./configure**, or with **make develop**.
- A reference counter named **Value::owner_count** in combination with class `Value_P`. A value is automatically deleted when its last owner (typically a **Token** is deleted.
- Member **Value::check_ptr** to detect double deletion or incorrect
- overwriting or double deletion of a value. Every **Value** constructor sets **check_ptr** to **(this ** 7)** which points into the middle of a **uint64** (of member `Value::shape`). The destructor later checks that **check_ptr** is still **(this ** 7)** (to detect overwriting) and clears it (to detect double deletion). The **check_ptr** is:
 - is different for every Value,
 - is a pointer that cannot occur otherwise,
 - becomes incorrect when a Value is incorrectly moved (e.g. with

memcpy()

Value_P is essentially a smart pointer (aka. shared pointer). A "normal" smart pointer typically consists of 2 pointers: * one pointing to the object itself, and * one to the reference counter that keeps track of the object's ownership (aka. the reference counter)

In contrast, the Value_P is a single pointer (to a Value), and the reference counter is allocated inside the value rather than somewhere else. The small advantage of a "normal" smart pointer is that they are not intrusive, which means that they can be used without any modifications of the object that they point to. But since we are in control of the Value class, this benefit does not really count and the smaller size of a Value_P outweighs the benefit of standard smart pointers. The Value_Ps are also decoupled from each other; one can always construct a Value_P from a Value * (and vice versa). The most important member functions of Value_P are:

- **Value_P()** : default constructor (points to no value)
- **Value_P(Value * val, const char * loc)** : construct Value_P from a Value *
- **Value_P(const Value_P & other, const char * loc)** : construct from **other**
- **Value_P(const Shape & sh, const char * loc)** : construct Value_P to a new value with shape **sh**. The ravel of that value is not (yet) initialized.
- **inline Value_P(const UCS_string & ucs, const char * loc)** : construct an APL text vector whose ravel Cells are initialized from the UCS_string,
- **inline Value_P(const UTF8_string & utf, const char * loc)** : construct an APL text vector whose ravel Cells are initialized from the UTF8_string.
ρValue is **utf.size()**, i.e. the UTF8 bytes are kept and no decoding into Unicodes is performed
- **void reset()** : decrement the owner-count and clear the pointer
- **init_pointer()** reset() without decrementing the owner-count
- **void clear()** : reset and add an event to the ValueHistory (if enabled)
- **bool operator +()** : **true** iff the pointer is valie (non-zero)
- **bool operator !()** : **true** iff the pointer is invalie (zero)
- **Value * operator→()** : return the Value * (to the Value)
- **Value & operator\()* :** return the Value & reference (of the Value)
- **void increment_owner_count()** : dito.
- **void decrement_owner_count()** : dito.
- **void isolate(const char * loc)** : if the value has more than one owner then clone (deep copy) the value and set pointer to the new copy.
- **void move(Value_P_Base & other, const char * loc)** :
 - release ownership of the current pointer,
 - take ownership of other's pointer, and
 - reset other's pointer. This operation does not change the owner_count

of the object.

All functions are inline and some can throw exceptions (primarily **WS FULL**). Note that some Value_P constructors above (e.g. *Value_P(const UCS_string & ucs, ...) initialize the entire Value (shape and ravel) while others (e.g. Value_P(const Shape & sh) only initialize the shape and leave the initialization of the ravel to the caller. In fact, the construction of a Value is always a two-step process (initializations of the shape followed by the initializations of the ravel) and the constructors that do both are mere convenience constructors for frequent cases. The following design pattern can found in many places:

```
Value_P Z(...); // construct Z (with un-initialized ravel)
    loop(x, z.element_count()) Z->next_ravel_XXX(...); // initialize the ravel
    Z->check_value(LOC) // set checked flag, detect un-initialized ravel items
    return Z;
```

The constructors for **Value*s are protected (with friend *Value_P)** to enforce that values are always constructed via **Value_P::Value_P(...)** rather than the corresponding **Value::Value(...)***. On the other hand, many functions use **const Value *** (**note the const!**) **instead of *Value_P**, primarily for passing function arguments. This is safe and slightly more efficient if the called function does not modify the Value (and hence its ownership) passed as argument.

Cloning of Values

The cloning (deep aka. recursive copy) of a Value is somewhat expensive in terms of performance. Cloning is, however, only needed if the clone is modified later (which would corrupt the original). When a **Value** is assigned to a Symbol (i.e. a variable) then it must be cloned because the variable may or may not be changed later. Think of a literal in a function body:

```
      ∇Z←FOO
Z←1 2 3
      ∇

X←FOO ◇ X←4 5 6
```

If we do not clone the result of FOO, not cloned before assigning it to X, then the subsequent assignment to X would modify the body of FOO and the next call of FOO would return 4 5 6.

On the other hand:

```
      ∇Z←FOO
Z←1 2 3
      ∇

X←1 + FOO ◇ X←4 5 6
```

is safe because the addition + does not modify the result of FOO but rather discards it immediately. GNU APL versions have always cloned values before assigning them (to be on the safe side). Current versions check how many owners

a value has before cloning them:

```
Value.hh

#define NEW_CLONE

#ifdef NEW_CLONE    /* new clone() scheme */

/// clone, given a Value_P. Result is a Value_P.
# define CLONE_P(B_P, L)    (B_P)

/// clone, given a const Value *. Result is a Value_P.
# define CLONE(pB, L)      Value_P(const_cast<Value *>(pB), L)

#else    /* old clone() scheme */

/// clone, given a Value_P. Result is a Value_P.
# define CLONE_P(B_P, L)    (B_P).get()->clone(L)

/// clone, given a Value *. Result is a Value_P.
# define CLONE(pB, L)      (pB)->clone(L)

#endif
```

The rules are:

- If a C++ function knows from its context that cloning is not needed (like in the second example above) then it may CLONE() and CLONE_P(). Determining if the cloning is actually performed, then depends on whether **NEW_CLONE** is **#defined** or not. This could be a mistake and then not **#define**'ing **NEW_CLONE** is a quick way to fall back to the old scheme.
- Otherwise, *in particular if **#define NEW_CLONE** makes a difference*, the function shall use Value→clone()* which clones unconditionally.

Ravel Cells

The ravel of every **Value** is a simple C array and not a C++ **std::vector<Cell>**. The alternative **std::vector<Cell>*** has some performance penalties when used for the ravel of a **Value**:

- destructor calls for every Cell when the vector<> is destructed, and
- no support for short values.

The other alternative, linked list of **Cells**, was also not an option due to their overhead in memory allocation (one **new()** **delete()** per Cell. The huge number of Cells in a typical APL program on the one hand and the typical operations performed with the Cells on the other result in the following, somewhat conflicting, requirements:

- fixed **Cell** size
- different cell contents (character, integer, float, complex, and nested) to support mixed APL values. In the old APL1 all values were homogeneous (and the type was a property of the value and its ravel). In APL2 the type became a property of the ravel Cells. Some APL interpreters optimize for the case of homogeneous ravels (probably a fair guess for APL2 programs

inherited from an APL1 code base) while GNU APL assumes mixed APL values as the default. Both assumptions (homogeneous vs. mixed) have advantages and drawbacks:

- Homogeneous arrays are faster to process, but this backfires when the automatic type conversion of APL (bool to integer, integer to float, float to complex takes place.
- Loops in the mixed case for scalar functions become much simpler because the different combinations of types (e.g. Boolean + Boolean, Boolean + integer, integer + integer, ...). Making e.g. dyadic **+(A, B)** a virtual member function **B.+(A)** of **B** and monadic **+(B)** a virtual member function **(B.+)** of **B**, removes two case statements (with the dyadic one being quite ugly).

The base class **Cell** therefore defines a pretty large number of virtual functions (a monadic and a dyadic one for each of the scalar APL functions:

`Cell.hh:`

```
...
virtual ErrorCode bif_ceiling(Cell * Z) const
virtual ErrorCode bif_conjugate(Cell * Z) const
virtual ErrorCode bif_direction(Cell * Z) const
...
virtual ErrorCode bif_add(Cell * Z, const Cell * A) const
virtual ErrorCode bif_and(Cell * Z, const Cell * A) const
virtual ErrorCode bif_and_bitwise(Cell * Z, const Cell * A) const
...
```

The base class **Cell** is inherited by the following derived classes:

- **CharCell**, their relevant content is a single 32-bit Unicode,
- **IntCell**, their relevant content is a single 64-bit signed integer,
- **RealCell**, their relevant content is a 64-bit floating point number,
- **ComplexCell**, their relevant content is two 64-bit floating point numbers
- **NumericCell**, the base class of **IntCell**, **RealCell**, and **ComplexCell**. Many Cell functions are almost identical for these base classes and these functions are implemented in **NumericCell** rather than implementing a copy in each base class. The simple trick that replaces hundreds of C/C++ **switch** statements and **case*s* is this:
 - base class **Cell** implements virtual functions that either throws a **DOMAIN ERROR** (e.g. for a character to integer conversion) or else return the desired type:
 - virtual Unicode get_char_value() const { DOMAIN_ERROR; }
 - virtual APL_Integer get_int_value() const { DOMAIN_ERROR; }
 - virtual APL_Float get_real_value() const { DOMAIN_ERROR; }
 - virtual APL_Float get_imag_value() const { DOMAIN_ERROR; }
 - virtual Value_P get_pointer_value() const { DOMAIN_ERROR }

- virtual `Cell * get_lval_value() const { LEFT_SYNTAX_ERROR; }`
- the derived classes then overload all access functions that they support. For example,
 - **CharCell** overload only **`get_char_value()`**
 - all **NumericCells** overload **`get_int_value()`**, **`get_real_value()`**, and **`get_imag_value()`** but not **`get_char_value()`**,
 - etc.
- **PointerCell**, their relevant content is a (nested) APL sub value.
- **LvalCell**, their relevant content is a **Cell** pointer. These **Cells** are used to implement "selective specification". A selective specification like **$(2p1 \downarrow A) \leftarrow B$** starts with (all **Cells** of) its the rightmost variable **A**, which is then more and more reduced (by **$1 \downarrow$** and **$2p$**) before the new value **B** can be assigned to the Cells that survive.

Instead of declaring the data members of Cells in the derived class (the normal approach, i.e. APL_Integers as member of IntCells, APL_Float members in FloatCells, etc.) they are declared as a union in the base Class Cell:

```
Cell.hh:
...
/// A union containing all possible cell values for the different Cell types
union SomeValue
{
    Unicode          aval;          ///< for CharCell
    APL_Float_Base   cval[2];       ///< for ComplexCell
    ErrorCode        eval;          ///< an error code
    APL_Integer      ival;          ///< for IntCell
    Cell             *lval;         ///< for LvalCell (selective assignment)
    ...
};
```

This union ensures that all derived classes have the same size.

Now back to the Cell functions. We take dyadic **$A + B$** and monadic **$+ B$** as an example, all other scalar APL functions follow suit:

1. Every scalar APL function is implemented in a separate associated class, derived from base **class ScalarFunction**. GNU APL uses the naming convention that the class name starts with **Bif_**, followed by the function arity (**F12_** for nomadic functions, **F2_** for purely dyadic functions).
2. The interpreter creates a single static instance of each function (such static instances are initialized by the compiler before `main()` is executed). The instance is always called **fun**, here **Bif_F12_PLUS::fun**. At the same time, a pointer **_fun** is created.
3. The single instance of the class is needed because the base class **Function** of class **ScalarFunction** has virtual functions (which is impossible for fully static classes). See [Functions](#) below for details.
4. The **eval_XXX()** functions like **Bif_F12_PLUS::eval_AB()** are implemented in inline function calls to **ScalarFunction::eval_scalar_AB()** which takes

the Cell function **Cell::bif_add** that distinguishes the behaviors of the (in total many) single instances of the classes derived from **ScalarFunction**.

5. The naming convention is that the virtual **Cell** functions is **Cell::bif_YYY**.
6. In short: base class **ScalarFunction** is essentially an iterator over the ravel the value arguments provided, and the difference between the derived functions are handled at the Cell level by means of different Cell functions.
7. While this may look a little complicated at the first glance, it is fairly efficient at runtime and saves a lot of mindless work for the GNU APL designer.
8. To summarize the example:

```
ScalarFunction.hh:
...
///
class Bif_F12_PLUS : public ScalarFunction
{
...
/// overloaded Function::eval_B().
virtual Token eval_B(Value_P B) const
    { return eval_scalar_B(B, &Cell::bif_conjugate); }

/// overloaded Function::eval_AB().
virtual Token eval_AB(Value_P A, Value_P B) const
    { return eval_scalar_AB(A, B,
        inverse ? &Cell::bif_add_inverse : &Cell::bif_add); }
...
static Bif_F12_PLUS * fun;           ///< Built-in function.
static Bif_F12_PLUS _fun;           ///< Built-in function.
};
```

Class IndexExpr

Class **IndexExpr** is a collection of 0 or more **Value_P**s or elided indices, where the **Value_P** 0 denotes an *elided index* (= all indices along an axis). **IndexExpr** objects are created and deleted by the prfix parser **Prefix.cc**. More precisely, a TOK_R_BRACK (for **]**), starts a new IndexExpr, a TOK_SEMICOL (for **;** append new values (axes) to an IndexExpr, and TOK_L_BRACK finalizes the creation of the **IndexExpr**. The next token (i.e. the token left of the **IndexExpr**) then calls the appropriate function (such as **Value::index()** or **Symbol::assign_indexed()** and deletes the **IndexExpr** again.

Classes Bif_XXX and Quad_XXX

APL offers a rather large number of primitives. In GNU APL, every primitive is implemented in its own class and according to the following rules:

- The class name has a prefix that indicates the arity of the primitive and is one of:
 - **Bif_F0** for niladic primitives; there is only one: **Bif_F0_ZILDE** (for \emptyset),
 - **Bif_F1_** for strictly monadic primitives; there is only one: ***Bif_F1_EXECUTE** (for \perp),

- **Bif_F2_** for strictly dyadic primitives,
- **Bif_F12_** for strictly nomadic primitives,
- **Bif_OPER1_** for monadic operators,
- **Bif_OPER2_** for dyadic operators, and
- **Quad_** for system functions. These are derived from class **QuadFunction**, primarily for grouping purposes. Note that the **Quad_** is also used for system variables. Similar to primitives, system functions with a large footprint have their own **.hh** and **.cc** files, while the remaining ones are collected in **QuadFunction.hh/QuadFunction.cc**.
- classes with a large code footprint have their own implementation files (followed by **.hh** or **.cc**), where the filename is the class name,
- the remaining classes (with a small code footprint) are declared/implemented in files **Primitivefunction.hh/Primitivefunction.cc**
- Every class is, directly or indirectly, derived from base class **Function**. **Class *FunctionC** defines all **virtual eval_XXX()** so that they throw a **VALENCE_ERROR**.
- Every class then overrides (only) those **virtual eval_XXX()** whose signatures it supports. For example, the strictly monadic primitive \pm (execute) overrides **virtual eval_B()**, the strictly dyadic primitive \square overrides **virtual eval_AB()**. Most primitives are nomadic and therefore override (at least) **virtual eval_B()** and **virtual eval_AB()**. With this technique arity mismatches between the runtime parser and the supported signatures of a function (for example: the parser calls the strictly dyadic function \neq monadically) are resolved without any runtime overhead.
- Due to the **virtual eval_XXX()** functions, every class needs a single (class static) instance which is called **_fun** and every such instance **fun** has a (static) pointer named **_fun** pointing to it.

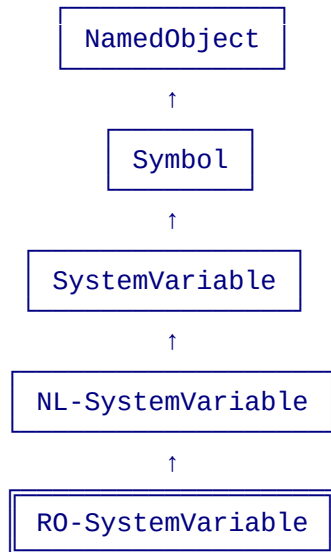
Except for the APL name (i.e. a single APL character for APL primitives and APL character \square followed by 0 or more letters for APL system functions) there is no major difference between APL primitives and APL system functions. The different names are handled in **Tokenizer.cc**; the other noteworthy difference is that *all APL primitives*, but only a rather small *subset of APL system functions* (and, for that matter, *system variables*) were standardized in the ISO standard for APL2.

Symbols

A symbol identifies an APL object created by the user, i.e. a defined function or a user defined variable. In GNU APL, system functions (\square -functions, e.g. \square FX) are *NOT* symbols but **APL primitives**. The commonality between \square -functions and APL primitives is larger than the similarity between \square -variables and \square -functions. The only noteworthy difference between APL primitives and \square -functions is their name and the fact that the APL primitives are defined in the APL standard, while most \square -functions and \square -variables are not. They are used by APL interpreter vendors to

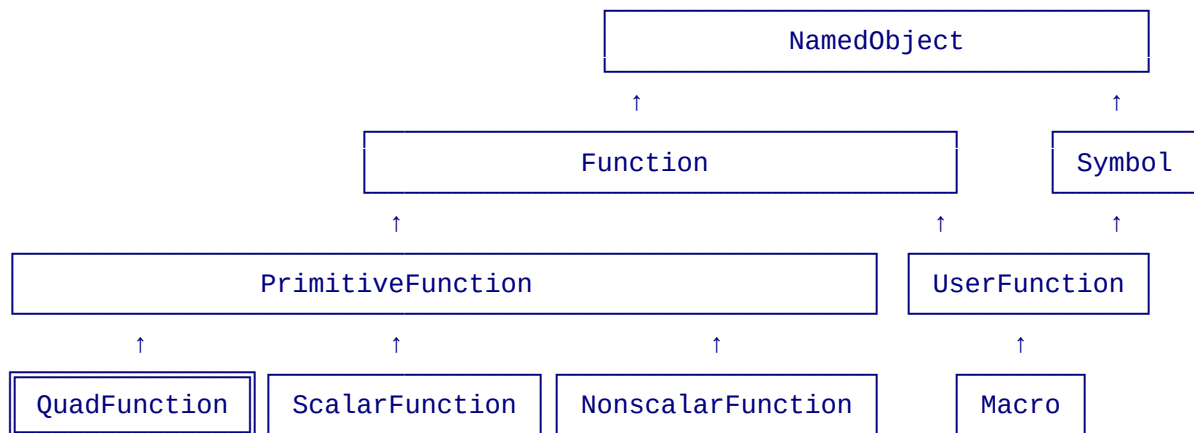
extend their functionality (and to lock-in their customers). The C++ class hierarchy in GNU APL (excerpts, the full picture is shown in the .png files) are:

html/classSystemVariable__inherit__graph.png:



and:

html/classFunction__inherit__graph.png:



Some explanations might be helpful:

- the common base class **NamedObject**:
 - provides **virtual UCS_string get_name()** to obtain the name of the object.
 - **virtual const Function * NamedObject::get_function()** returns a valid **Function *** for functions, otherwise 0.
 - **virtual Symbol * NamedObject::get_symbol()** returns a valid **Symbol *** for symbols, otherwise 0.
- **SystemVariables** are variables like `□IO`, `□FC`, ...
- **NL_SystemVariables** are **Non-Local** system variables (for which localizing is permitted, but has no effect): `□L`, `□R`, ...

- **RO_SystemVariables** are **Read-Only** global system variables (for which localizing makes no sense): `□AI`, `□ARG`, ...
- base class **Function** defines default behavior of (**virtual eval_B()**, **eval_XB()**, **eval_AB()**, **eval_AXB()** ...). The default behavior is to throw a **VALENCE ERROR**, and derived classes simply overload the `eval_XXX()` functions that they implement.
- **QuadFunction**, **ScalarFunction**, and **NonscalarFunction** are groupings that simplify the implementation. For example, all scalar functions are implemented in the same way and only differ by their details at the Cell level.
- **UserFunctions** are the only functions that are also Symbols (so that their name appears in the symbol table. The system functions get their name via their ID.
- **Macro** is a **UserFunction** that is defined and can only be used by the interpreter itself. It cannot be called from APL programs and simplifies built-in operators when they are called with **UserFunction** arguments considerably.

Symbols are created when the tokenizer detects a name for the first time **SymbolTable::lookup_symbol()**, **SymbolTable::lookup_existing_symbol()**, and **SymbolTable::add_symbol()**. Every symbol has a stack **Symbol::value_stack** of **ValueStackItems**, which is pushed when the symbol is localized (at the beginning of a defined function) and popped when the execution of the defined function ends. The **ValueStackItem** at the top of the **Symbol::value_stack** defines the role (scope) that the symbol currently has. The possibilities are:

NamedObject.hh:

```
enum NameClass
{
    NC_INVALID           = 0x0100,    ///< invalid name class.
    NC_UNUSED_USER_NAME = 0x0200,    ///< unused user name, not yet assigned
    NC_LABEL             = 0x0401,    ///< Label.
    NC_VARIABLE          = 0x0802,    ///< (assigned) variable.
    NC_FUNCTION          = 0x1003,    ///< (defined) function.
    NC_OPERATOR          = 0x2004,    ///< (defined) operator.
    NC_SYSTEM_VAR        = 0x4005,    ///< system variable.
    NC_SYSTEM_FUN        = 0x8006,    ///< system function.
    NC_case_mask         = 0x00FF,    ///< almost □NC
    NC_bool_mask         = 0xFF00,    ///< for fast selection

    NC_FUN_OPER          = (NC_FUNCTION | NC_OPERATOR) & NC_bool_mask,
    NC_left              = (NC_VARIABLE   |
                          NC_UNUSED_USER_NAME |
                          NC_SYSTEM_VAR   |
                          NC_INVALID      // □, □, □xx
                          ) & NC_bool_mask
};
```

The **NameClass** is equally well suited for C/C++ **switch** statements (by using the lower byte) and for testing groups of **NameClasses** (by using the upper byte). After the tokenization (or localization) of a symbol it has the NameClass **NC_UNUSED_USER_NAME**; this changes to e.g. **NC_VARIABLE** when a value is assigned to the symbol, or to **NC_FUNCTION** when a defined function with the

name of the symbol is defined with `□FX` or the `∇`-editor.

The actual role of a user defined symbol can change over time, while the role of a system name is fixed and cannot be changed. E.g. `□IO` is always a variable and `□UCS` is always a function.

Classes `SymbolTable` and `SystemSymTab`

Every Symbol is contained in one of two symbol tables:

- User-defined symbols are stored in a static instance of class `SymbolTable`, while
- system defined symbols (for system functions and system variables) are stored in a static instance of class `SystemSymTab`.

Both symbol tables are contained in the base class `Workspace_0` of the static instance `Workspace::the_workspace`. The classes `SymbolTable` and `SystemSymTab` have a lot in common, but also differences. The common functionality is implemented in their template base class `SymbolTableBase`:

`SymbolTable.hh`:

```
...
/// common part of user-defined names and distinguished names
template <typename T, size_t SYMBOL_COUNT>
class SymbolTableBase
{
    ...
};
```

The differences are:

- the table size (256 entries for `SystemSymTab` and 65536 entries for user defined names (constant `SYMBOL_HASH_TABLE_SIZE` in file `SystemLimits.def`
- On success the lookup of a user defined name returns a `Symbol *`, while a lookup of a system name returns a `SystemName *`. Symbols are somewhat more complicated than `SystemName` because their role (like function or variable) can change over time while every `SystemName` has a fixed role (either function or variable) cannot. Also, every `SystemName` has a unique Id (defined in `Id.def`) while *all* user defined symbols have the same Id named `ID_USER_SYMBOL`.
- In a way `SystemNames` are more lightweight versions of `Symbols`.
- Symbol lookup. A lookup function has a single `UCS_string` argument with the name of a symbol. It returns a valid pointer on success and 0 otherwise.
 - The base class `SymbolTableBase` has a lookup function `SymbolTableBase::lookup_existing_symbol()` which returns a valid (non-zero) pointer.
 - The derived class `SymbolTable` has another lookup function `SymbolTable::lookup_symbol()` which always succeeds. Instead of returning 0 to indicate that a given symbol name is not in the table, it

creates a new symbol with the name given.

- In contrast, class **SystemSymTab** has functions **add_function()** and **add_variable()** to add system functions and system variables (whose role is, unlike for user defined names, implied by their Id).
- Both tables use a hash function **SymbolTableBase::function compute_hash()** to determine the position of a name in the table, from which a linked list of colliding names starts.
- Erased names are not removed from a symbol table, but rather marked erased in the Symbol itself, while the symbol name remains in the table. This is to avoid searching of a user defined symbol in all existing tokens. Also, a symbol name could be)ERASEd at the top level but still exist as the name of a local in some defined function.

The lookup (and possibly insertion of a new) Symbol uses the following algorithm (**SymbolTable::lookup_symbol(const UCS_string & sym_name)**):

1. compute a hash value from **sym_name**
2. if **symbol_table[hash]** is **0** then the name is the first symbol with that hash (unsuccessful lookup case 1.)
 1. Create a **new Symbol** with the name **sym_name**,
 2. Store the new symbol in **symbol_table[hash]**, and
 3. return. the new symbol.
3. Otherwise one or more symbols with the same hash already exists and the name may or may not already exist in the symbol table.
 1. Chase the linked list from **symbol_table[hash]** to its end (via pointer **Symbol::next**).
 2. If a symbol with name **sym_name** is detected on the way then return that symbol (successful lookup).
4. Otherwise **sym_name** was not in the table. Chase the linked list again.
 1. If a symbol is detected on the way with:
 1. an empty value stack, or
 2. a value stack with 1 item and name class **NC_UNUSED_USER_NAME**,
 3. then re-initialize that symbol with a new name **sym_name** and return the symbol (unsuccessful lookup case 2). The condition 4.a.i. or ii. is returned by **Symbol::is_erased()**.
 - Otherwise (unsuccessful lookup case 3), the linked list contains no erased symbols that could be re-used. Append a new Symbol at the end of a list, like in (unsuccessful lookup case 1.) above.

This algorithm

- avoids unnecessary creation of new symbols by re-using formerly erased

symbols (step 4.a.i above), and

- avoids unnecessary removals and insertions into the linked list by not removing symbols that are (temporarily) not used, e.g. after an)ERASE command.

The case 4.a.i is not (or no longer) supposed to happen since all Symbol constructors now **push()** an item with name class **NC_UNUSED_USER_NAME** onto the Symbol's value stack and the **pop()** function asserts if the last item of the value stack would be removed,

The primary user of the symbol tables is the tokenizer (which translates human readable names into **Symbols**. The vast majority of symbol table lookups occur when APL code is tokenized (▽-editor, □FX) along with only a few system functions (those that use variable or function names or APL code strings at runtime).

Executables

Every APL input line is first tokenized, and the result of this tokenization is an **Executable**. The primary content of any **Executable** is a **TokenString** named **body**:

Token.hh:

```
class Token_string : public std::vector<Token> { ... }
```

Executable.hh:

```
// Base class for ExecuteList, StatementList, and UserFunction
class Executable
{
    ...
    Token_string body;
    ...
}
```

The rest of class **Executable** are helper functions that support the execution of the raw **body**.

There are 3 cases where **Executables** are produced (in order of decreasing relevance):

1. The definition of a defined function, either interactively with the ▽-editor, or programmatically in APL with □FX.
2. The evaluation of a single input line in immediate execution, and
3. The evaluation of an APL string with APL primitive ⊞ (aka. Execute).

For each of these 3 cases (which also correspond to the **ParseMode** mentioned above), a corresponding class for that case is derived:

Executable.hh:

```
...
class UserFunction : public Function, public Executable { ... }
class StatementList : public Executable { ... }
class ExecuteList : public Executable { ... }
....
```

In some debug outputs, as well as in some commands and in □SI, the derived class of a particular **Executable** is displayed as a single character like this:

- class UserFunction: ∇
- class StatementList: ◇
- class ExecuteList: ⊥

Each of a derived class has a static function **fix()** which creates an instance of that class. The **fix()** function acts as a constructor for the derived class. Unlike a normal constructor (which either returns a valid object of the class or else throws an exception, the **fix()** function returns 0 if the construction fails (typically due to some SYNTAX ERROR in the input line(s) passed to it. The **fix()** function of class **UserFunction** is essentially the implementation of **monadic □FX**, and is also called after the ∇-editor finalizes a function definition (trailing ∇).

The main differences between the 3 classes are:

Derived Class	Number of lines	Statements/ Line	Jumps/ Labels	Arguments	Result
ExecuteList	single line	one	no	no	yes
StatementList	single line	several	no	no	no
UserFunction ∇, □FX	multiple lines	several	yes	yes	yes
UserFunction { ... }	single line	one	no	yes	yes

Classes ExecuteList and StatementList

The execution of **ExecuteLists** and **StatementLists** is rather simple: the tokens of their body are interpreted from the start of the body to its end, no arguments can be passed to the body, no change of the execution with branches (→N), etc..

Class UserFunction

Defined functions (class UserFunction) are far more complex. As can be seen above, class UserFunction inherits from the two classes **Function** and **Executable**.

The inheritance from **Function** overrides its various virtual **eval_XXX()** functions, so that a defined function can be called like an APL primitive, The arity of a defined function (monadic, dyadic, operator, etc.) is determined by the function header and since class UserFunction cannot know it before the header is parsed, it must overload all possible virtual **eval_XXX()** declared in base class **Function**.

The inheritance from **Executable** provides the construction of the body (tokenization of the function lines (in class **Tokenizer**), pre-parsing (in class **Parser**), and some optimizations (also in class **Parser**).

There are two aspects of class **UserFunction** that are not considered in either of its base classes:

- Local variables, and
- Branches ($\rightarrow N$)

These aspects are implemented by means of two member variables *

vector<Function_PC> UserFunction::line_starts is a jump table that tells, for every function line N , which token is the start of line N in the (linear) body of the executable. For APL jump statement, say $\rightarrow N$, **line_starts[N]** is the first token in line N and its execution simply continues at token **body[line_starts[N]**. * when a defined function is called then its arguments (if any) are assigned to the formal arguments declared in the function header and the local variables in the header are pushed. This happens in member **UserFunction::UserFunction_header header**, which is a functional grouping of the somewhat large class **UserFunction**.

Let:

```
∇Z←A SUM B;C;D
  Z←A + B
∇
```

The typical life-cycle of Userfunction SUM is then:

1. The **Userfunction** named **SUM**, is created with the ∇ -editor or with **□FX** (which calls **Userfunction::fix()**).
2. The APL pattern matching (**Prefix.cc**) detects a call of **SUM**, say pattern **3 SUM 42**.
3. The function is called:
 1. Symbols **Z**, **A**, **B**, **C**, and **D** are pushed (and become, at this point in time, temporarily undefined).
 2. The actual arguments **2** and **42** of **SUM** are assigned to the formal arguments **A** and **B** of **SUM**. that is, **A←3** \diamond **B←42**; **C** and **D** remain unassigned.
 3. The **Executable::body** is executed. This computes **Z←45**.
4. At some point in time **SUM** returns (either when the end of the body is reached, or the when execution is stopped explitley (e.g. $\rightarrow 0$)).
 1. The current value of **Z** (if any) is set aside (and will later be returned to the caller).
 2. The Symbols **Z**, **A**, **B**, **C**, and **D** are popped again (restored).
 3. The pattern matched above (**3 FOO 42**) is replaced by former value of **Z** (i.e. **45**).

Class StateIndicator

An **Executable**, as described above, is a static object that is essentially the

tokenization of some APL source code string(s). An Executable is created once and deleted if no longer needed. Between the creation and the deletion of an Executable it remains unchanged (aka. static). An Executable is therefore comparable to the object code (binary) of a compiled language.

To produce a result, an Executable must be *executed*. While the Executable itself remains unchanged, the different executions of the same Executable can have rather different results, e.g. an APL value in one execution and an error in another.

Every execution of an Executable is performed in a context which tracks the state and the progress of the execution. This context is an object of class **StateIndicator**. The complete execution of an Executable consists of the following principle steps:

1. Creation of a new StateIndicator, then
2. Execution of the Tokens in the body of the executable, then
3. returning the result of the execution (if any) to the initiator of the execution, and finally
4. Destruction of the StateIndicator.

In step 2. above, the execution of (the same or a different) executable is started, i.e. before the current execution has completed. To accomplish this, the different contexts of different executions, regardless of whether the execution concerns the same Executable (direct recursion) or different ones (function call, possibly indirect recursion), are organized as a stack, known as the *)SI stack*. In GNU APL the)SI stack is a singly-linked list of StateIndicator objects, the link between adjacent StateIndicator objects in the)SI stack is member

StateIndicator::parent.

The constructor of StateIndicator has two arguments: the executable to be executed and the parent that has initiated the execution:

StateIndicator.hh:

```
...  
/// constructor  
StateIndicator(const Executable * exec, StateIndicator * _par);  
...
```

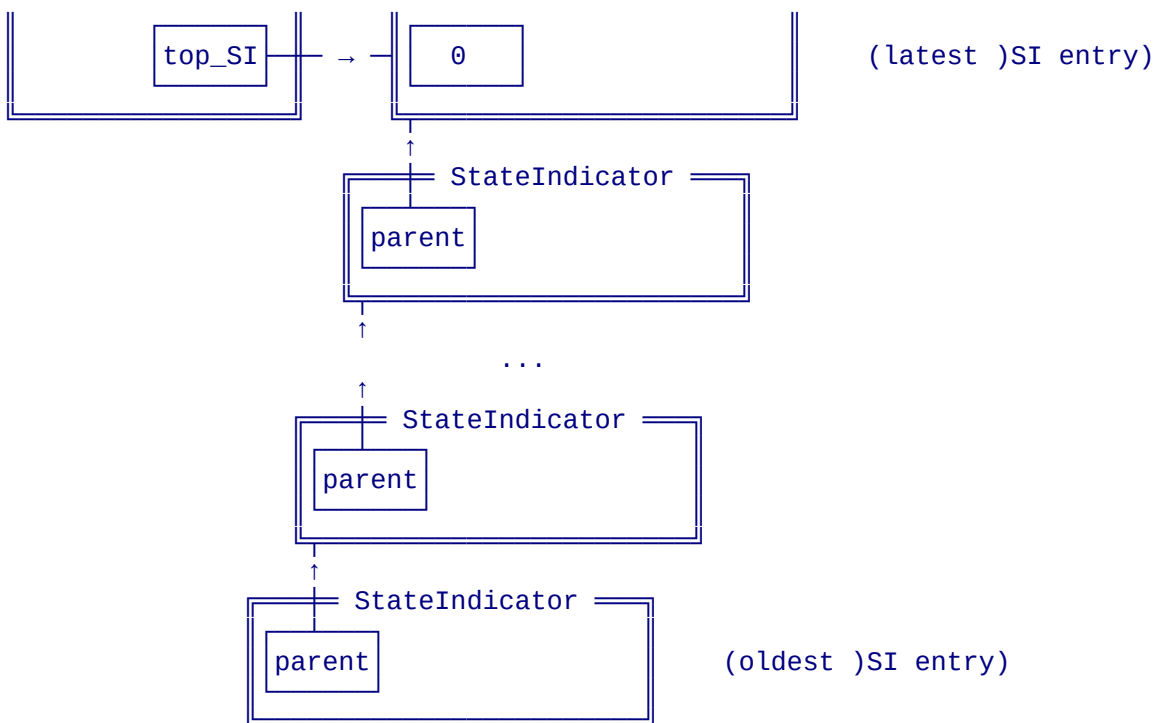
The constructor initializes **StateIndicator::parent**, which does not change thereafter. Since the)SI stack is a linked list, it can be represented as a single pointer to the most recent StateIndicator object. This pointer is stored in class **Workspace**:

Workspace.hh:

```
...  
/// the SI stack. Initially top_SI is 0 (empty stack)  
StateIndicator * top_SI;
```

The full picture is something like this:

◻◻◻ Workspace ◻◻◻ ◻◻◻ StateIndicator ◻◻◻



By and large the only active StateIndicator is the topmost one. The StateIndicators below it are relatively passive and only come back to life when the topmost StateIndicator, sometimes referred to as **TOS** (for **Top Of Stack** has finalized step 4. above. An error in the execution of TOS normally does not end it, but pushed another StateIndicator (with mode immediate execution) onto the stack. The only cases that remove (pop) one or more items from the SI stack are:

- the executable in TOS reaches the end of the body of its Executable. This is the normal case,
- the user issues)SIC command (in immediate execution mode). This clears the entire)SI stack,
- the executable in TOS performs a \rightarrow statement (aka. Escape), which rolls back the stack until the pop'ed StateIndicator was an Executable with mode immediate execution. That Executable was the one pushed above (i.e. due to an error). This pops the topmost item(s) until the TOS is an immediate execution context,
- the execution of TOS had an error and TOS was under the control of \square EA, \square EB, or \square EC. This pops the topmost item(s) until TOS has the \square EA, \square EB, or \square EC that catches the error (as opposed to the normal error handling which would have pushed another StateIndicator onto the stack rather than popping items of it).

The TOS is needed in many different situations, therefore class **Workspace** provides, for convenience, a number of static wrapper functions that access frequently used members of its **top SI**.



Strictly speaking the *APL)SI stack* refers to the entire linked list of contexts, i.e. to **Workspace::top_SI**. In GNU APL, though, the StateIndicator class implements an individual entry in that linked list.

The Class has two more members of interest:

```

StateIndicator.hh:
...
/// details of the last error in this context.
Error error;

/// the current-stack of this context.
Prefix current_stack;
...

```

StateIndicator::error contains the error information (if any) for the StateIndicator, e.g. for □EM. There can be several StateIndicators on the stack that were (temporarily) suspended and can be continued later (by some →N back into the failed Executable and therefore information on several errors may be present in the different StateIndicator objects.

StateIndicator::current_stack is far more interesting. It manages the inner loop that runs along the body of the Executable of the StateIndicator. See below.

Class Prefix

Class **Prefix** is an iterator over the body of an Executable. Its most important member is a stack named **content**:

```

Prefix.hh:

...
/// put pointer (for the next token at PC). Since content is a stack,
/// its put position is also its size.
int put;

/** the lookahead tokens (tokens that were shifted but not yet reduced).
    \b content is in body order, that is, content[0] is the oldest
    (= rightmost in APL order) token and content[put - 1] is the latest.
    */
Token_loc content[MAX_CONTENT];
...
/// one phrase in the phrase table
struct Phrase
{
    const char *    phrase_name;        ///< phrase name
    const char *    reduce_name;        ///< reduce function name
    void (Prefix::*reduce_fun)();      ///< reduce function
    unsigned int    phrase_hash;        ///< phrase hash
    int             prio;               ///< phrase priority
    int             misc;               ///< 1 if MISC phrase
    int             phrase_len;         ///< phrase length
    uint8_t         sub_nodes[TC_MAX_PHRASE+1];  ///< parent nodes
};

```

Member **Prefix::content** is by and large processed as follows:

1. The constructor **Prefix::Prefix()** starts with an empty stack.
2. If the stack starts with a valid phrase, then:
 1. The function **Phrase::reduce_fun** in the phrase is called;
 2. The **reduce_fun()** is a non-static member of class **Prefix** and can therefore access the content in order to produce a result. The

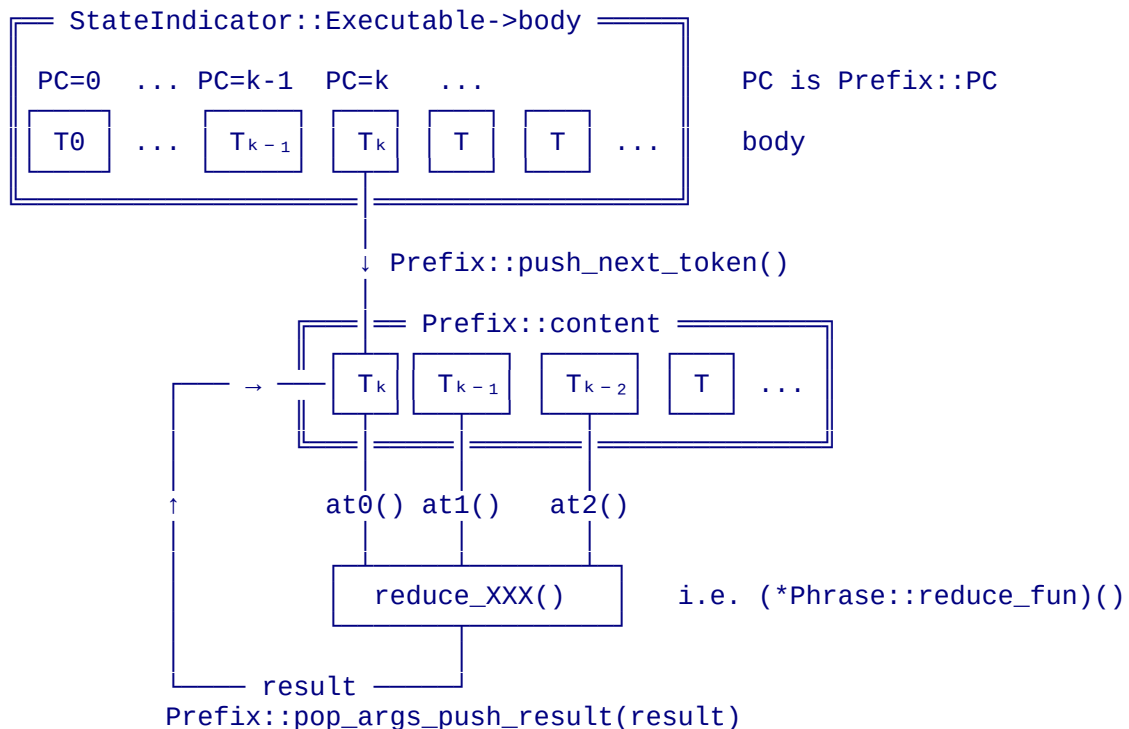
reduce_fun() has no arguments because accessing **content** directly is far more efficient than using (C++) function arguments.

3. The **reduce_fun()** will then:

1. extract the relevant data from the tokens **Prefix::content**, remove (pop) the tokens from the **Prefix::content**, and
2. push the result (if any) onto the **Prefix::content**.

3. Otherwise the stack does not start with a valid phrase. In this case the next **Token_loc** is fetched from the body of the **Executable**.

The following picture may illustrate the above (the entire algorithm is somewhat more complicated):



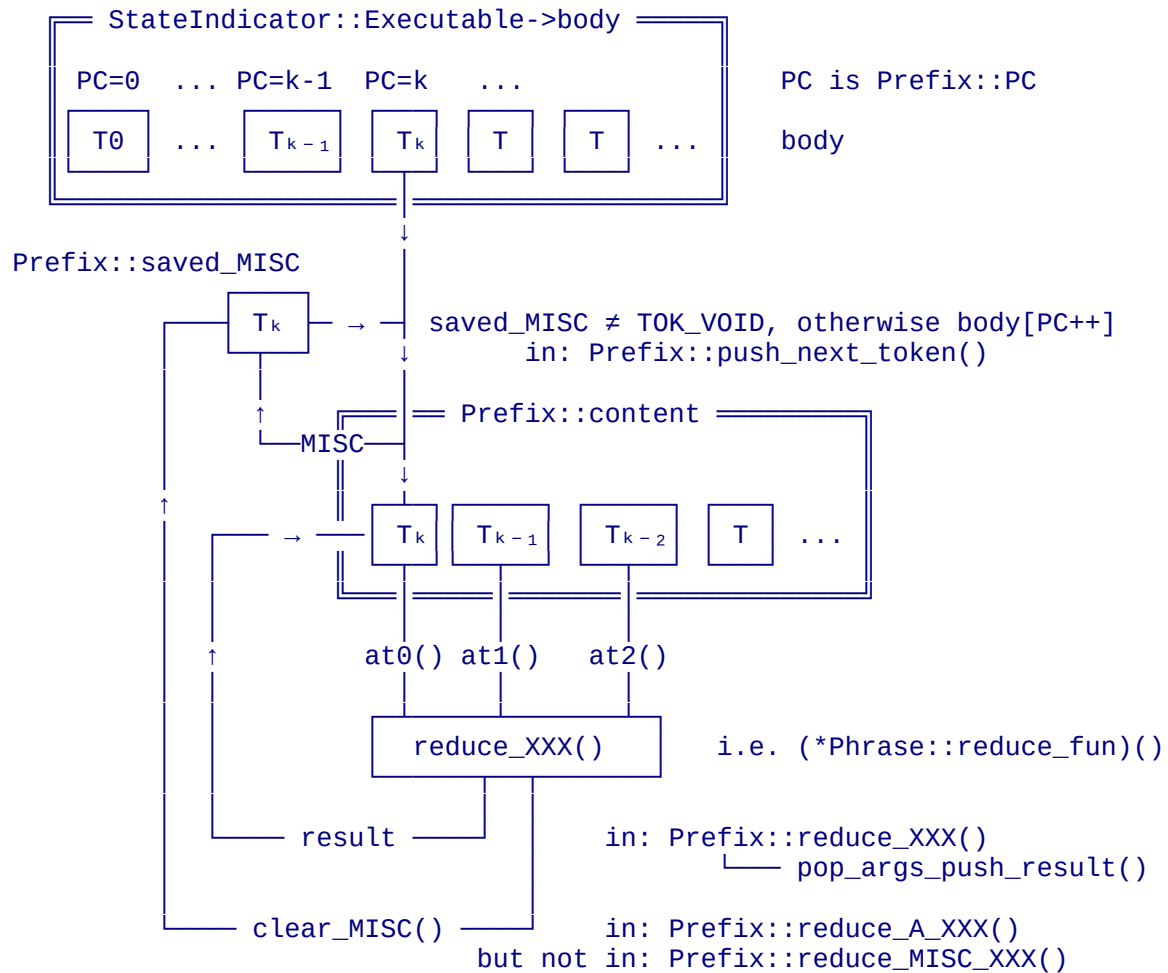
Apart from **Prefix::content**, Prefix has the following members of interest (the others just helpers that make the code more readable):

- The Functions **Prefix::at0()**, **Prefix::at1()**, **Prefix::at2()**, and **Prefix::at3()** are primarily used by the different **Phrase::reduce_funs** in order to access the (up to) 4 leftmost items in **Prefix::content**. The phrases have different lengths (from 1 to 4) and the matching of a phrase guarantees that sufficiently many **Token_locs** exist on the stack. It is important to note that, although the body of the **Executable** is in reversed order as explained earlier, **at0()**, **at1()**, **at2()**, **at3()** are in APL order (to make the code easier to understand). For example the items in phrase **A + B** are:
 - **at0()** \leftrightarrow **A**,
 - **at1()** \leftrightarrow **+**, and
 - **at2()** \leftrightarrow **B**.
- Put differently, **Prefix::content**, has a "hot end" that is manipulated with

Prefix::push_next_token() and the various reduce_XXX() functions and a "cold end" containing tokens that are put on ice for a while.

- **Prefix::PC** is the **P**rogram **C**ounter. The APL interpreter can be considered as a (lightweight) virtual machine whose opcodes are Tokens and whose program(s) are lists of Tokens (= **Executables**). In this model **PC** is the program counter of the virtual machine.
 - The execution of an executable starts with PC=0 (= start of the body) in constructor **Prefix::Prefix()**.
 - Whenever a new **Token_loc** is needed (in Step 3. above) then the new **Token_loc** is: **Token_loc(body[PC], PC)** and PC is incremented.
 - The execution is complete when **PC == body.size()** (or after **→0** in defined functions).
- **saved_MISC**: many APL functions are *nomadic*, which means that they can be called monadically or dyadically. This includes *most* APL primitives and *all* non-niladic defined functions.
 - There is a fixed set of **MISC** tokens (more precisely: **TokenClasses**) that cause a nomadic function to be evaluated monadically: $\leftarrow \rightarrow [\diamond F C M$ and $($.
 - All phrases that start with a MISC token (and differ only by that token) use the same **Phrase::reduce_fun**. For example, the two APL statements **X←|1.5** and **-|1.5** will both first match phrase **MISC_F_B**. This is because
 - (aka. F) and \leftarrow are MISC tokens. Therefore both phrases will first compute **|1.5** in **Prefix::reduce_MISC_F_B()** and result **1**.
 - In contrast, in APL statement **2|1.5** the leftmost value **2** is NOT a MISC token and therefore **Prefix::reduce_A_F_B()** will be called.
 - Unfortunately the decision about whether **Prefix::reduce_MISC_F_B()** or **Prefix::reduce_A_F_B()** matches can only be made *after* either **MISC** or **A** was fetched. IOW: this decision requires a lookahead of 1 token (and Prefix is therefore essentially a so-called LALR(1) parser). If the decision is in favor of **Prefix::reduce_A_F_B()**, then everything is fine. However, for **Prefix::reduce_MISC_F_B()** the MISC token is not used and must be considered somehow. The solution is to store the MISC token in C++ variable **Prefix::saved_MISC**. Note that one cannot simply decrement the PC in order to *undo* the fetching of the MISC token. Decrementing the PC looks much simpler at first glance, but would only work if the MISC token came directly from the body of the **Executable**. Sometimes, though, (defined function **F**, defined operator **M**, or complete index **C**) the MISC token is the result of a Symbol resolution (cases **F** and **M**) or even of a prior computation with side effects (case complete index **C**).

The following picture is a slightly more detailed version of the previous one:



APL Functions

APL knows two types of functions:

- APL primitives (functions and operators) and system functions. They are defined by the interpreter itself and are implemented in C++, and
- defined functions, They are defined by the APL programmer.

Every function has a **signature** that defines the combination of arguments that the function can be called with. The total number of possible signatures is somewhat large, but a single function only implements a rather small subset of them and returns a **VALENCE ERROR** when the **Prefix** parser calls a function with a signature that the called function does not support. In GNU APL the signature is a bitmap of **signature atoms** where every atom corresponds to a possible position of an argument in a **Prefix** pattern. However, not all possible bitmaps are valid signatures (e.g. a function with a left argument but no right argument):

```
APL_types.hh:
...
/// signature of a defined function
enum Fun_signature
{
    // signature atoms
    //
    SIG_NONE          = 0,          ///<
    SIG_Z              = 0x01,      ///< function has a result
    SIG_A              = 0x02,      ///< function has a left argument
}
```

```

SIG_LO          = 0x04,    ///< operator left operand
SIG_FUN         = 0x08,    ///< function (always set)
SIG_RO         = 0x10,    ///< operator right operand
SIG_X           = 0x20,    ///< RO has an axis
SIG_B           = 0x40,    ///< function has a right argument

// operator variants
//
SIG_FUN_X       = SIG_FUN | SIG_X,    ///< function with axis
SIG_OP1         = SIG_LO | SIG_FUN,    ///< monadic operator
SIG_OP1_X       = SIG_OP1 | SIG_X,    ///< monadic operator with axis
SIG_OP2         = SIG_OP1 | SIG_RO,    ///< dyadic operator
SIG_LORO        = SIG_LO | SIG_RO,    ///< monadic or dyadic operator

// argument variants
//
SIG_NIL         = SIG_NONE,            ///< niladic function
SIG_MON         = SIG_B,               ///< monadic function or operator
SIG_DYA         = SIG_A | SIG_B,       ///< dyadic function or operator

// allowed combinations of operator variants and argument variants...

// niladic
//
SIG_F0          = SIG_FUN | SIG_NIL,    ///< dito

SIG_Z_F0        = SIG_Z | SIG_F0,       ///< dito

// monadic
//
SIG_F1_B        = SIG_MON | SIG_FUN,    ///< dito
SIG_F1_X_B      = SIG_MON | SIG_FUN_X,  ///< dito
SIG_LO_OP1_B    = SIG_MON | SIG_OP1,    ///< dito
SIG_LO_OP1_X_B  = SIG_MON | SIG_OP1_X,  ///< dito
SIG_LO_OP2_RO_B = SIG_MON | SIG_OP2,    ///< dito

SIG_Z_F1_B      = SIG_Z | SIG_F1_B,     ///< dito
SIG_Z_F1_X_B    = SIG_Z | SIG_F1_X_B,   ///< dito
SIG_Z_LO_OP1_B  = SIG_Z | SIG_LO_OP1_B, ///< dito
SIG_Z_LO_OP1_X_B = SIG_Z | SIG_LO_OP1_X_B, ///< dito
SIG_Z_LO_OP2_RO_B = SIG_Z | SIG_LO_OP2_RO_B, ///< dito

// dyadic
//
SIG_A_F2_B      = SIG_DYA | SIG_FUN,    ///< dito
SIG_A_F2_X_B    = SIG_DYA | SIG_FUN_X,  ///< dito
SIG_A_LO_OP1_B  = SIG_DYA | SIG_OP1,    ///< dito
SIG_A_LO_OP1_X_B = SIG_DYA | SIG_OP1_X,  ///< dito
SIG_A_LO_OP2_RO_B = SIG_DYA | SIG_OP2,  ///< dito

SIG_Z_A_F2_B    = SIG_Z | SIG_A_F2_B,   ///< dito
SIG_Z_A_F2_X_B  = SIG_Z | SIG_A_F2_X_B,  ///< dito
SIG_Z_A_LO_OP1_B = SIG_Z | SIG_A_LO_OP1_B, ///< dito
SIG_Z_A_LO_OP1_X_B = SIG_Z | SIG_A_LO_OP1_X_B, ///< dito
SIG_Z_A_LO_OP2_RO_B = SIG_Z | SIG_A_LO_OP2_RO_B, ///< dito
};
...

```

There are $0x80 = 128$ possible signature bitmaps of which 22 are valid.

The base class **Function** of all functions has a non-virtual function **get_signature()** which is computed from some virtual functions related to signature atoms:

```

Function.cc:
...

```

```

un_signature
Function::get_signature() const
{
int sig = SIG_FUN;
    if (has_result())    sig |= SIG_Z;
    if (has_axis())      sig |= SIG_X;

    if (get_oper_valence() == 2)    sig |= SIG_R0;
    if (get_oper_valence() >= 1)    sig |= SIG_L0;

    if (get_fun_valence() == 2)     sig |= SIG_A;
    if (get_fun_valence() >= 1)     sig |= SIG_B;

    return Fun_signature(sig);
}

```

System functions do not use signatures because their classes know the signatures that they support already at compile time. Defined functions are different because all are handled by the same class **UserFunction** even though the signatures of different functions are defined by the user in the function header and the signatures differ. The only class that cares about the signature (of its left function argument) is **Bif_OPER1_EACH**.

For every signature **XXX** that a function class supports it must overload the corresponding **eval_XXX()** function. In our example, primitive **+** supports monadic **+B** and dyadic **A+B**. It therefore overloads **eval_B()** and **eval_AB()**. Likewise the axis variant **A[X]** is overloaded with **eval_AB()** and then delegated to their companions in class **ScalarFunction**.

```

ScalarFunction.hh:
...
class Bif_F12_PLUS : public ScalarFunction
{
...
virtual Token eval_B(Value_P B) const
    { return eval_scalar_B(B, &Cell::bif_conjugate); }

/// overloaded Function::eval_AB().
virtual Token eval_AB(Value_P A, Value_P B) const
    { return eval_scalar_AB(A, B,
        inverse ? &Cell::bif_add_inverse : &Cell::bif_add); }
...
}

```



You may wonder what is the matter with **bif_add_inverse()**. Some primitives have an inverse function which, if present, can be used by the dyadic operator ***** (power operator, not to be confused with the power function *****) for negative power arguments. The inverse of **A+B** is **A-B** because $A = (A+B) - B$. Most of the circular functions "**A○B**" have inverses but most others do not. The power operator is, for good reasons, neither defined in the ISO APL standard, nor in the IBM APL2 language reference, but implemented in GNU APL. Avoid it where possible.

To summarize the execution of APL functions:

1. The Prefix parser matches a pattern, for example **A F B**.
2. The reduce function for the pattern **A F B** is **reduce_A_F_B()**.
3. The **Function *** in token **F** is e.g. **Bif_F12_PLUS::fun**.

4. The pattern also implies that the `eval_AB()` function of **F**, i.e. `Bif_F12_PLUS::fun→eval_AB(A, B)` shall be called in order to produce the reduction result.
5. **Bif_F12_PLUS::fun→eval_AB()** is inlined and calls **ScalarFunction::eval_AB()** with argument **Cell::bif_add**. It returns some result Z.
6. The Prefix parser replaces **A F B** with **Z** which finalizes the execution of **A F B**.

All other derivatives of class **Function** are executed in the same fashion.

Important differences

An important difference between **ExecuteList**, **StatementList**, and **UserFunction** concerns the last token in their body. This token is not produced by the tokenizer (since there is no APL code for it), but by the corresponding **fix()** function of the derived class. The last token determines if and how a value is returned to the caller of the executable:

Executable Type	Final Token	Result
ExecuteList	TOK_RETURN_EXEC	APL Value
StatementList	TOK_RETURN_STATS	None
UserFunction w/o result	TOK_RETURN_VOID	None
UserFunction with result	TOK_RETURN_SYMBOL	APL Value (if assigned)

Class Nable

Class **Nable** implements the interactive function editor (for lack of a better name, sometimes referred to as the ∇-editor). The constructor **Nable::Nabla(const UCS_string & cmd)** starts the editor with the first editor command (which either creates a new function or else modifies an existing one). Every serious APL programmer knows the ∇-editor and its implementation is straightforward.

IBM APL2 defines the ∇-editor to be recursive (so one can start the editing of some other defined function while editing another defined function), this feature was not implemented in the GNU APL editor.

Class Workspace

Class **Workspace** is a container that includes all objects that were created or modified by the user (through the execution of APL programs, APL commands, and/or APL expressions). These objects are:

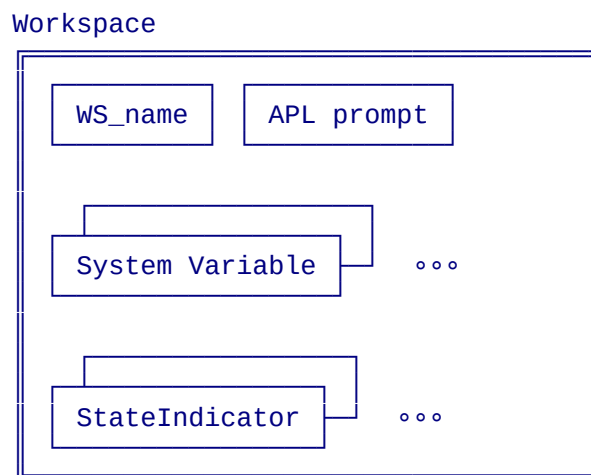
- Defined functions,
- User variables,
- (The values of) System variables,

- the symbol table, and
- The state indicator, aka. *the SI stack*.

APL primitives are constant in nature therefore need not be stored in a workspace.

The APL interpreter has (only) one static **Workspace** instance named **Workspace::the_workspace**.

The **)SAVE** command serializes the binary **Workspace::the_workspace** into a human readable .xml file. The **)LOAD** or **)COPY** commands may later de-serialized a .xml back into the **Workspace::the_workspace** of another APL session. Many of the APL commands modify the state of **Workspace::the_workspace**. According to **Workspace.hh** the data relevant members of class **Workspace** are:



As a matter of fact, class **Workspace** is derived from a simple container **Workspace_0** which contains 2 more items **symbol_table** and **distinguished_names**,

Workspace.hh:

```

class Workspace_0
{
protected:
    /// the symbol table for user-defined names of this workspace.
    SymbolTable symbol_table;

    /// the symbol table for system names (aka. distinguished names) of
    /// this workspace.
    SystemSymTab distinguished_names;
};
  
```

The idea between this distribution between base class **Workspace_0** and derived class **Workspace** is twofold:

1. It ensures that the two symbol tables **symbol_table** (for user-defined names) and **distinguished_names** for system functions and system variables are properly initialized (to empty tables) so that the constructor of the derived class **Workspace** can simply add system functions and variables without taking care of draining the tables first.
2. It separates the interpreter state produced by the user (and which is **)SAVE**

into a .xml file) from the interpreter state produced by the initialization of the interpreter itself. Since *system functions* are constant in nature, they have no state related to them. Therefore the initial state of the interpreter is primarily the state of all system variables, for example after a **)CLEAR** command.

Class XML_Archive

The **)SAVE**ing or **)LOAD**ing of workspaces into or from resp. .xml files is a somewhat lengthy process. It therefore has its own implementation files **Archive.hh** and **Archive.cc**. Class **Workspace** is the only user of these classes. There are actually 3 classes:

- Class **XML_Archive** is the base class. It merely contains an **enum ArchiveSyntax** for verifying that the archive software versions of a **)LOAD**ed .xml file is compatible with the software version that **)SAVE**d the archive (and warns the user if not).
- Derived class **XML_Saving_Archive** contains the functions for serializing a **Workspace** object into a file. Serializing means that the internal (binary) representation of a workspace is written into a (human readable) format from which it can later be de-serialized. IOW: class derived **XML_Saving_Archive** implements the **)SAVE** command.
- Class **XML_Loading_Archive** contains the functions for de-serializing a **Workspace** object from the (human readable) format back to its internal (binary) representation. IOW: class **XML_Loading_Archive** implements the various **)LOAD** and **)COPY** commands for .xml files.

Serialization (APL to XML)

The serialization is split into a small number of categories or *loops*. Each such loop iterates over the same kind of data structures in the workspace. To simplify the later de-serialization of a .xml file, the order of the different loops matters, while the order of items inside a loop is arbitrary. The loops are:

1. loop over all defined functions, then
2. loop over all APL values, then
3. loop over all symbols, then
4. loop over user defined commands (a non-standard GNU APL feature), then
5. loop over **)SI** stack entries.

The de-serializer expects the XML tags to appear in the same order. The following table shows the mapping between C++ types and XML tags (some trivial C++ types were omitted):

C++ type	XML Tag(s) and attributes
Cell []	<Ravel vid="..." bytes="..."/>
Prefix	<Parser size="..." assign-pending="..." action="..." lookahead-

C++ type	XML Tag(s) and attributes
	high="..."> ... </Parser>
StateIndicator	<SI-entry level="..." pc="..." line="..."> Body </SI-entry>
Symbol	<Symbol name="..." stack-size="..."> Value Stack... </Symbol>
SymbolTable []	<SymbolTable size="..."> Symbols... </SymbolTable>
Token	<Token pc="..." tag="..." val="..." />
UserFunction	<Function vid="..." /> Body </Function>
UserCommand []	<Commands size="..."> Command... </Commands>
UserCommand	<Command name="..." mode="..." fun="..." /> rk="..." sh_0="...".../>
UCS_string	<UCS uni="..." />
Value	<Value flg="..." vid="..." parent="..."

Some remarks:

- The XML tag corresponds to the C++ type,
- the tag attributes correspond to the data members of the C++ type. To the extent possible, all data members are stored in a single tag (i.e. <TAG ... /> in XML) as opposed to start and end tags (i.e. <TAG ...> ... </TAG> in XML. The exceptions are C++ classes of variable size (arrays, vectors, linked lists, etc.), whose items are serialized between the corresponding start and end tags.
- empty vectors etc. may be omitted

Most C++ types are mapped to XML in a straightforward fashion. The exceptions are described in more detail in the following.

Serialization of APL Values

The serialization of APL Values is implemented with two different loops: one for the (typically large) ravel of the value, and one for the rest (i.e. shape, flags, etc.

In C++, objects refer to each other by their addresses in the memory (aka. *pointers* in C, and/or *references* in C++). Even though one could, in theory, emulate pointers with file offsets, this would lead to a rather complicated implementation. Instead, GNU APL uses IDs that define relations between different objects in the serialization; the IDs thus replace pointers and/or references in C++.

The IDs are different **enum** types (so that the C++ compiler can detect the use of incompatible pointers). The **enum Vid { ... }** is the ID that relates XML <Value .../> tags to the corresponding XML <Ravel .../> tag.

For example, Let:

```
)CLEAR
)WSID TEST

Z←1 2 3

)SAVE
```

The)SAVE command serializes workspace TEST (into file TEST.xml) like this:

TEST.xml:

```
...
<Ravel vid="15" cells="³1³2³3"/>
...
<Value flg="0x400" vid="15" parent="-1" rk="1" sh-0="3"/>
...
<SymbolTable size="1">
  <Symbol name="Z" stack-size="1">
    <Variable vid="15"/>
  </Symbol>
</SymbolTable>
```

The serialization of the <Ravel> deserves some more explanation. As already explained, the **vid="15"** attribute of the <Ravel> tag links the ravel to the <Value with the (same) **vid="15"**. This information alone suffices to de-serialize the right side **1 2 3** of the assignment **Z←1 2 3**. However, the assignment to APL variable (aka. Symbol) **Z** has created a second tag **<Variable vid="15"/>** which links the (current) APL value **1 2 3** to symbol **Z**. In the above example **Z** is a global variable, therefore there is only one **<Variable ...>** tag between the **<Symbol> ... </Symbol>** tags. However, if the symbol **Z** were localized, then every localization would have added a new XML item between the **<Symbol> ... </Symbol>** tags.

Now, a "proper" XML would have serialized the ravel **1 2 3** as follows:

```
<Ravel vid="15"> <Cell type="integer" value="1"/> <Cell type="integer"
value="2"/> <Cell type="integer" value="3"/> </Ravel>
```

Since ravels tend to be rather long (so that Cells are frequent), the "proper" <Cell /> tags were optimized into a far more compact format which may be called **micro tags** (or **µ-tags**). A micro tag is a single Unicode character that defines the following properties and purposes that are related to (and later needed to de-serialize) a Cell:

- the C++ Cell Type (CharCell, IntCell, FloatCell, ComplexCell, PointerCell, or LvalCell)
- the Encoding used for the Cell value (only relevant for CharCell), and
- to serve as a delimiter between adjacent Cells of the ravel.

The different encodings of CharCells are needed because XML has restrictions on the characters that are permitted in XML attributes and to abbreviate the most likely characters (i.e ASCII).

The following table lists the micro tags used:

μ-Tag	C++ Cell Type	Encoding	Example	Cells
⁰	CharCell	end of Unicode character sequence	² Hello ⁰	≥ 1
¹	CharCell	sprintf("%X", ASCII-value)	¹ A (aka. LF)	1
²	CharCell	start of Unicode character sequence	² World ⁰	≥ 1
³	IntCell	integer (sequence of digits 0-9)	³ 42	1
⁴	FloatCell	sprintf("%.17g", value)	⁴ 42.56	1
⁵	ComplexCell	sprintf("%17gJ%17g", real, imag)	⁵ 1.2J3.4	1
⁶	PointerCell	integer (vid)	⁶ 42	1
⁷	LvalCell	sprintf("%d[%lld]", vid, offset)	⁷ 42[2]	1
⁸	FloatCell	integer ÷ integer	⁸ 2÷3	1

All Cell types except CharCell use one tag per Cell and the Cell value extends from the first character after the μ-Tag to the character before the next μ-Tag. Since the number of ASCII characters in the Cell value is somewhat larger, the overhead for the μ-Tag is relatively small.

In contrast, CharCells contain one character and the μ-tags tags for a string like "Hello" would require 5 bytes for the ASCII "Hello", plus 10 to 15 bytes for the 5 μ-Tags (since the Unicodes of the μ-Tags are UTF8-encoded, and most have 2 or 3 bytes per Unicode).

Therefore, and since APL characters more often occur in groups than as individual characters, the ²...⁰ values can span entire strings with only one start μ-Tag ² and one end μ-Tag ⁰. This is similar to APL where *Hello* is the same as *H e l l o*.

Serialization of UCS_strings

UCS_strings do not only appear as groups of characters in the ravel of APL values (as described above) but also in other contexts (e.g. the APL text that defines a defined function). For these cases, a common XML tag **<UCS uni="...">** is used; the value of the tag attribute **uni=** is encoded with μ-Tags ⁰, ¹, and ² as described above. For example. the APL function:

```

▽Z←A DIV B
  Z←A÷B
▽

```

will be serialized as:

```

<Symbol name="DIV" stack-size="1">
  <Function fid="0x55BCF671DC80" creation-time="1687101368729013"
    exec-properties="0,0,0,0">
    <UCS uni="²Z←A DIV B⁰¹A
      ² Z←A÷B⁰¹A

```

```

        ">
    </Function>
</Symbol>

```

Note the **1A** above, which is the ASCII line feed character **0x0A** (which is not allowed in a tag attribute value in XML).

De-Serialization (XML to APL)

All the complexity above was motivated by the following design goals:

- **SAVE**d workspaces should be reasonably compact,
- it should be possible to fix a broken workspace with a simple ASCII text editor such as vi or emacs, with only some basic knowledge of the XML syntax, and
- the **LOAD**ing and **COPY**ing of workspaces (i.e. the decoding of XML files should be simple and, for portability and maintenance reasons, not depend on external libraries.

Almost every serialization function, say **save_XXX()**, in class **SavingArchive** has a corresponding de-serialization function **read_XXX()** in class **LoadingArchive**:

SavingArchive	LoadingArchive	C++ class (de-)serialized
save()	read_Workspace()	entire Workspace
save_Derived()	read_Derived()	DerivedFunction
save_functions()	-	UserFunction (all)
save_Function()	read_Function()	UserFunction (one)
save_Function_name()	read_Function_name()	(helper for UserFunction)
save_Parser()	read_Parser()	Prefix
save_Ravel()	read_Ravel()	Cell (all in a ravel)
save_shape()	-	Shape
save_SI_entry()	read_SI_entry()	StateIndicator
save_Symbol()	read_Symbol()	Symbol
save_symtab()	read_SymbolTable()	SymbolTable
save_token_loc()	read_Token()	Token
save_UCS()	read_UCS()	UCS_string
save_user_commands()	read_Commands()	Command::user_command []

SavingArchive

save_vstack_item()

LoadingArchive

-

C++ class (de-)serialized

(helper for Symbol)

De-serialization (aka, decoding) is inherently more complex than serialization (aka, encoding). Therefore some more **read_XXX()** functions have no corresponding **save_XXX()** function.

Parallel (Multi-Core) Execution

Parallel execution of APL has a longer history.

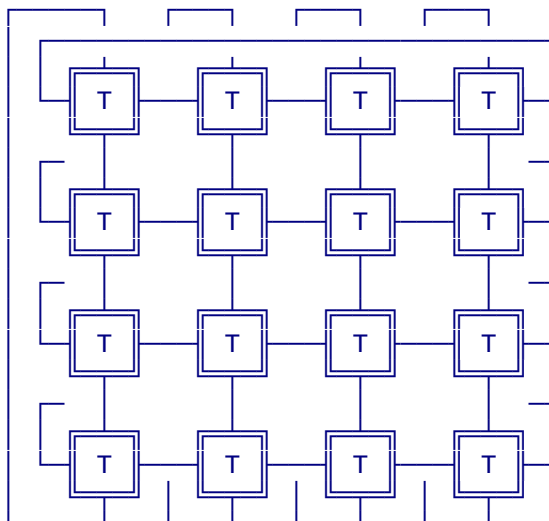
Background

In the early 1980s the author learned APL (after ALGOL, Assembler, and BASIC, and before C. C++ was in its early days and not (yet) commercially available. The platform was an IBM 5110 desktop, a precursor to the IBM PC which came later. C compilers and APL interpreters were expensive and the IBM 5110 was slow, particularly in comparison with Assembler or compiled languages like C on comparable platforms.

Shortly after the owner of the IBM 5110 replaced it with a faster 16-bit machine (a Motorola 68000 CPU with an APL 68000 interpreter (today named APL-X) and multi-user capabilities). APL performance was still an issue.

To address the performance issue, the author undertook the construction of a parallel APL computer. The first draft architecture was based on **Transputers**, a brand new CPU technology at that time (which died out shortly after). The advantage of Transputers was that every CPU had 4 fast communication channels which could be connected to neighboring Transputers. The resulting topology is then a torus (a mesh with the leftmost Transputers connected to the corresponding rightmost Transputers and the top Transputers connected to the bottom Transputers):

4x4 Transputer Torus

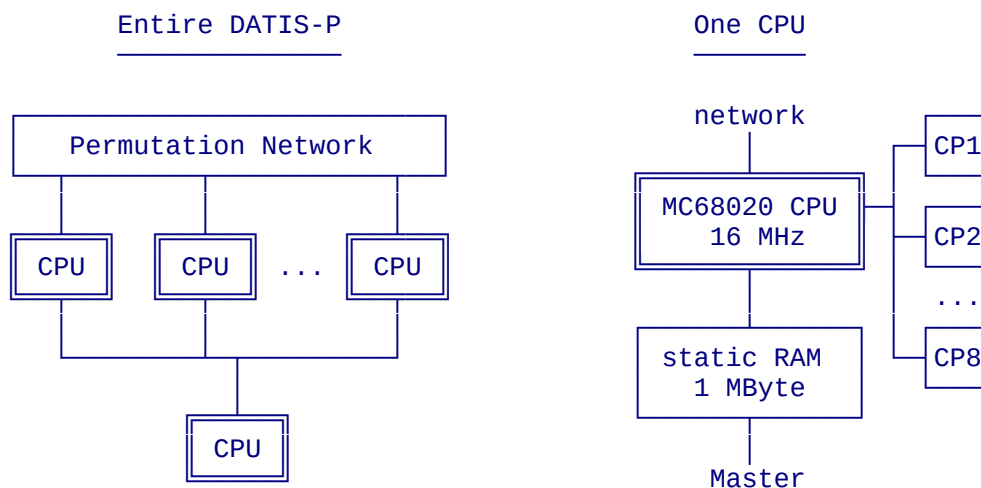


Another architecture that was initially considered was the Intel Hypercube.

However, neither the Transputer torus nor the Intel Hypercube seemed to suit APL very well. We therefore decided to develop our own architecture named DATIS-P (derived from the first names of the students that developed the hardware and software developed for the DATIS-P and probably the last name of the author. The developers were (in DATIS-P order): Dieter Scheerer [Sch], Andreas Gergen [Ger], Thomas Mörsdorf [Mör], and last but not least Ina Gläs [Glä]. The DATIS-P was prepared for 256 CPUs, but for budgetary reasons our prototype had only 32 CPUs. The power consumption was about 50 Amperes at 5 Volts (which blew the fuses in our lab when we turned it on for the first time). The size was three 19-inch cabinets (roughly the size of a mainframe at that time).

The most challenging part of the DATIS-P was its communication network. Its structure was a 3-stage permutation network as invented by **Waksman**. Since no suitable technology existed at that time we developed an ASIC (a 1-bit wide 8×8 crossbar; the network for 32 CPUs was 8-bit wide and was comprized of $8\text{-bit} \times 3\text{ stages} \times 4\text{ width} = 96$ such ASICs. The 3 stages were pipelined, a reconfiguration of the network to an arbitrary permutation of the inputs was done in parallel by all CPUs and took only 3 CPU instructions. The communication bandwidth was 2 CPU instructions per byte (one MOVE.B instruction to write into the network and one MOVE.B to read the byte (written by some other CPU) back. The total bandwidth of the network of our 32-CPU machine was therefore $(\text{a } 16\text{ MHz clock}) \div (3\text{ clocks per MOVE.B}) \div (2\text{ MOVE.B per transfer}) = 2.6\text{ MByte/s}$ or 21 MBit/s per CPU = 670 MBit/s for the entire network (a lot at that time). The bandwidth of the DATIS-P scales linearly with the number of CPUs, therefore a 256 CPU machine would have had a total communication bandwidth of 5.3 Gigabit/s.

The architecture of the DATIS-P is:



An entire DATIS-P consists of:

- The permutation network,
- 32 worker CPUs, and
- one master CPU. This master CPU does the APL interpretation and delegates all ravel computations to the worker CPUs. To support scalar extensions like **1 + 2 3 4 5 ...** the master was able to broadcast the scalar 1 into the dual port memories of the worker with a single instruction.

Every CPU of the DATIS-P:

- was a Motorola 68020 processor (the cutting edge when the development of DATIS-P started),
- had up to 8 numerical co-processors MC 68881 to boost the floating point performance
- A dual-port 1 MByte static memory; one port connected to the local CPU the other to the Master CPU.

The development of the hardware (PCB designs, ASIC design) and the software took from 1985 to 1989; the author was managing the hardware development and finished a Ph.D thesis about the parallel algorithms for the DATIS-P in 1990 [\[Sau\]](#).

The primary conclusion of the thesis was:

On the DATIS-P architecture with P processors:

- All scalar APL functions can be computed in parallel with a speed-up of $O(P)$. That is, the performance of every scalar function scales linearly.
 - Most of the non-scalar APL functions can be computed in parallel with speed-up $O(P \div \log P)$. That is, the other functions scale almost linearly.
 - The remaining non-scalar functions are those that require sorting. Their speed-up is $O(P \div \log^2 P)$ (using Batcher's Bitonic Sort).
 - There is one exception which may happen for particular index vectors X in the indexed assignment $A[X] \leftarrow B$, in the indexed reference $A \leftarrow B[X]$, and functions derived from them, e.g. X/B . If the index vector X is such that all ravel items of A are concentrated on the same worker CPU, then the speed-up can degrade down to $O(1)$ because that worker CPU has to do all of the work while the others are idle. Usually, though, the speed-up is $O(P \div \log^2 P)$ i.e. like the sorting functions Δ and ∇ . These days one can use P-RAM machines that do not suffer from this problem, but in 1990 P-RAMs were not available.
-

The benchmarks performed on the DATIS-P essentially confirmed the predictions of the Ph.D. thesis. Since we had no APL interpreter at that time, we only implemented the most important parallel algorithms performed by the workers. The implementation languages were C on the master and Assembler on the workers.

These results were also submitted to the 1990 APL conference in Copenhagen, Denmark. Without any feedback. The author therefore pursued a career in the telecommunications industry.

Around 2000, the author came across a PC demo version of IBM APL2 and played a little with it. However, the license cost for a full version was entirely out of range for a private user. At about the same time he worked with Unicode, UTF8-

encodings and the like and discovered that all APL characters were available in Unicode. Probably at that time the idea of writing my own APL interpreter began to mature.

Around 2008, the first version of the interpreter was ready. It was not yet a GNU project and not very stable either.

In 2013 the interpreter was accepted by the GNU project and became GNU APL. Version 1.0 was sort of stable (i.e. it did not crash immediately), but still rather buggy (primarily suffering from memory leaks).

In 2014, multi-core support (parallel APL) was added. However, only the APL workhorses (i.e. scalar functions and their outer products) were parallelized and benchmarked. The results were somewhat disappointing and suggested that non-scalar functions will not gain much by parallelization. They are memory bound (as opposed to the more compute bound scalar functions) The main reason seems to be that the interface between the CPU cores and the memory does not scale with the number of cores so that the cores are competing for memory access. In contrast, the DATIS-P had only local memories whose total bandwidth grows linearly with the number of CPUs.

Implementation



Parallel APL is a purely experimental feature that should not be used for production code. Expect crashes from time to time.

The details are described in a separate document [Parallel-APL.html](#), which is shipped with GNU APL in the same directory as this document.

Exercise 1

We conclude our journey through the GNU APL source code with a little exercise for the alert reader:

*Add a new monadic system function named \square **FOO** to GNU APL. \square **FOO** takes a 1-item integer argument **B** and returns the APL value (string) "FOO-B".*

You may look backwards in this document and consult the GNU APL source code, but not peek into the solution sketch below. If you stumble over something that is not properly described above then please report that on <mailto:bug-apl@gnu.org>.

Solution (sketch with comments)

1. Add a symbol **FOO** to **src/Id.def**:

```
qf( FOO , " $\square$ FOO" , )
```

The proper position of **FOO** in **Id.def** is important (after the last \square -symbol group that starts with **F**, i.e. after the \square **FFT** line.

2. Add a header file **src/Quad_FOO.hh** like this:

```

#ifndef __Quad_F00_HH_DEFINED__
#define __Quad_F00_HH_DEFINED__

#include "QuadFunction.hh"
...

class Quad_F00 : public QuadFunction
{
public:
    /// Constructor.
    Quad_F00()
        : QuadFunction(TOK_Quad_F00),
          system_wisdom_loaded(false)
    {}

    static Quad_F00 * fun;          ///< Built-in function.
    static Quad_F00 _fun;          ///< Built-in function.

protected:
    /// overloaded Function::eval_B()
    Token eval_B(Value_P B) const;

    ...
};

#endif // __Quad_F00_HH_DEFINED__

```

Since `□FOO` is monadic, we have one handler `eval_B(Value_P B)`, which overloads **`virtual Function::eval_B()`** in **`Function.hh`** and causes our **`Quad_F00::eval_B`** to be called instead of **`eval_B()`** of the base class. If `□FOO` should support more signatures then it is better to add them at a later point in time.

3. Add a C++ source file **`src/Quad_F00.cc`** like this:

```

#include "Quad_F00.hh"

Quad_F00 Quad_F00::_fun;
Quad_F00 * Quad_F00::fun = &Quad_F00::_fun;

//-----
Token
Quad_F00::eval_B(Value_P B) const
{
    Value_P Z(LOC);                // makes Z an APL scalar,
    Z->next_ravel_int(42);          // which is the integer 42
    Z->check_value(LOC);            // check Z and set its Z->VF_complete
    flag
    return Token(TOK_APL_VALUE1, Z); // return the result.
}
//-----

```

We have not (yet) implemented the requested functionality in **`Quad_F00::eval_B()`**. It makes life easier to first integrate `□FOO` into the GNU APL framework (and test the integration) then to add the functionality later on.

4. When using SVN then add the new files to the repository:

```

$ svn add Quad_F00.cc Quad_F00.hh
$ svn commit -m "new file"

```

5. Add the following line to several source files:

```
#include "Quad_F00.hh"
```

This makes the new declarations available in the **.cc** files that need it. The files concerned are:

1. src/Archive.cc
2. src/Id.cc
3. src/QuadFunction.cc
4. src/Workspace.cc

6. Update **Makefile.am**.

1. Add the new files **Quad_F00.cc** and **Quad_F00.hh** to the make variable **common_SOURCES** in **src/Makefile.am**:

```
Quad_F00.cc Quad_F00.hh
```

keep **src/Makefile.am** sorted alphabetically and indented in the same way as the other files.

2. (re-)run **autoreconf** (in the GNU APL top-level directory)

```
$ autoreconf
```

This creates a new **src/Makefile.in** from **src/Makefile.am**, and

3. (re-)run **./configure** (also in the GNU APL top-level directory).

```
$ ./configure
```

This includes the new files in **src/Makefile** so that they are properly compiled and linked.

7. Add a **TD()** macro for **□FOO** in **src/Token.def**:

```
TD(TOK_Quad_F00      , TC_FUN1      , TV_FUN  , ID::Quad_F00      )
```

This defines a new Token with token class **TC_FUN1**, value type **TV_FUN** i.e. **Function** (-pointer), and ID **ID::Quad_F00**. Note that:

1. **TC_FUN1** is an alias for **TC_FUN12** (and so is **TC_FUN2**. These aliases are used to make explicit that **□FOO** is monadic. If you should add a dyadic signature later on then change the token class to **TC_FUN12**.
2. Since **src/Token.def** is a **.def** file, it will be **#inluded** by several **.cc**

files and magically insert the new token in several source files.

8. Add a **sf_def()** macro to **src/SystemVariable.def**:

```
sf_def(Quad_F00,    "F00",    "Definitely not □BAR"    )
```

This takes care of a number of things:

1. **)SAVE** of functions in **.xml** workspace snapshots,
2. Tab expansion of □-function names,
3. update of the **)HELP** command (short help text), and
4. Instantiation in workspaces.

9. And finally: increment **ASX_MINOR** minor in **enum ArchiveSyntax** in file **src/Archive.hh**. Adding □-functions to GNU is a backward compatible change of the **.xml** file format. The new interpreter can **)LOAD** files that were **)SAVE**d with older versions, but not vice versa. Incrementing **ASX_MINOR** will then issue a warning in older (but not too old) interpreters.

10. After these changes you can build and test that the new interpreter compiles and works (if not then the author has missed something above).

```
$ make
$ src/apl

□F00 5
42
```

11. If so, then we can add some more beef into **Quad_F00::eval_B()**:

```
#include "Quad_F00.hh"

Quad_F00 Quad_F00::_fun;
Quad_F00 * Quad_F00::fun = &Quad_F00::_fun;

//-----
Token
Quad_F00::eval_B(Value_P B) const
{
    // check that B is a 1-item integer
    //
    if (B->element_count() != 1)    // rank or length error
    {
        if (B->get_ranke() != 1)    RANK_ERROR;
        else                        LENGTH_ERROR;
    }

    const Cell & B0 = B->get_cfirst();    // first item in B
    const APL_Integer b = B0.get_int_value();    // DOMAIN_ERROR for non integers
    UCS_string z("F00-");    // z is string "F00-"
    z.append_number(b);    // now z is e.g. "F00-42"

    Value_P Z(z);    // constructor: APL value from string
    return Token(TOK_APL_VALUE1, Z);    // return the result.
}
//-----
```

Exercise 2

This exercise is similar to [\[Exercise 1\]](#) above, but this time we want to create a system variable instead of a system function:

Add a new system variable named `□FOO` to GNU APL so that:

- *`*□FOO←B*` with 1-item integer B shall remember B , while*
- *`*Z←□FOO*` shall be the APL value (string) `"FOO-B"`.*

System variables are inherently more complex than system functions because most system functions are stateless, while system variables typically have state that needs to be **)SAVE**d in workspaces.

Solution (sketch with comments)

The solution is for the most part, similar to the previous example. We therefore only discuss the differences.

- System variables are (directly or indirectly) derived from class **SystemVariable**, while system functions are derived from class **Function**. System functions overload one or more (virtual) **eval_XXX()**, while system variables overload the following (virtual) functions:
 - **Value_P get_apl_value() const**, called when the variable is referenced,
 - **assign(Value_P B, ...)**, called when a values is assigned to the variable,
 - possibly **assign_indexed()**, called for indexed assignments (if the variable shall support it), and
 - possibly **push()** and **pop()** functions if the variable can be localized.
- Therefore our **Quad_FOO.hh** reads:

```
//-----
#ifndef __Quad_FOO_HH_DEFINED__
#define __Quad_FOO_HH_DEFINED__

/**
 * System variable Quad-FOO.
 */
/// The class implementing □FOO
class Quad_FOO : public SystemVariable
{
public:
    /// Constructor.
    Quad_IO()
    : SystemVariable(ID_Quad_IO)
    {
        Symbol::assign(IntScalar(0, LOC), false, LOC);
    }

protected:
    /// overloaded Symbol::assign().
    virtual void assign(Value_P B, bool clone, const char * loc);

    // overloaded Symbol::push()
```

```

virtual void push()
{
    Symbol::push();
    Symbol::assign(IntScalar(0, LOC), false, LOC);
}
};

#endif // __Quad_F00_HH_DEFINED__

```

- and the **TD()** macro now becomes:

```
TD(TOK_Quad_F00, TC_SYMBOL, TV_SYM, ID_Quad_F00)
```

- The **sf_def()** macro (**sf** stands for **s**ystem **f**unction) above in **SystemVariable.def** now becomes:

```
rw_sv_def(Quad_F00, "F00", "not BAR ")
```

where **rw_sv_def** stands for **r**ead/**w**rite **s**ystem **v**ariable. If **FOO** were read-only (which implies that localizing it has no effect), then macro **ro_sv_def** would have been used instead.

The name **SystemVariable.def** is a little misleading because it contains the macros for both variables and functions.

References

- [ISO] ISO/IEC Standard 13751. *Programming Language APL, Extended*
www.math.uwaterloo.ca/~ljdickey/apl-rep/docs/is13751.pdf



This file is, despite its **.pdf** extension, a **gzip** compressed file which will only expand into a proper **.pdf** file after uncompressing it (with **gunzip**).

- [IBM] IBM Corp. *APL2: Language Reference*
www.ibm.com/downloads/cas/ZOKMYKOY.



The exact download location seems to change every now and then.

- [Sau] Dr. Jürgen Sauermann *Ein Paralleler APL Rechner (A Parallel APL Computer)*.

Ph.D Thesis 1990, Universität des Saarlandes.

- [Sch] Dipl. inf. Dieter Scheerer *Entwurf und Realisierung eines Verbindungsnetzwerks als Teil des Multiprozessorsystems DATIS-P-256*.

Masters Thesis 1989, Universität des Saarlandes.

- [Ger] Dipl. inf. Andreas Gergen *Entwurf und Realisierung einer Master-Slave*

Kommunikation im Multiprozessorsystem DATIS-P-256

Masters Thesis 1989, Universität des Saarlandes.

- [Mör] Dipl. inf. Thomas Mörsdorf *Entwurf, Bau, und Test einer 32-bit CPU als Teil des Multiprozessorsystems DATIS-P-256.*

Masters Thesis 1989, Universität des Saarlandes.

- [Glä] Dipl. inf. Ina Gläs *Die Lösung spezieller Probleme bei der Realisierung eines Permutationsnetzwerkes als Teil des Multiprozessorsystems DATIS-P-256.*

Masters Thesis 1989, Universität des Saarlandes.