
The ATerm Programming Guide

Table of Contents

Introduction	1
ATerms at a glance	3
The ATerm data type	3
Operations on ATerms	4
Using the C ATerm Library	9
Initializing and using the ATerm library	9
Memory Management of ATerms	10
ATerm formats	11
Level One Interface of C ATerm Library	13
Level One Types	13
A note on `blobs' and BAF	14
Level One Functionality	14
Level Two Interface of C ATerm Library	22
Level Two Types	22
Level Two Functionality	23
Command Line Utilities	36
ATerm-conversion: baffle	36
Calculating the size of an ATerm: termsize	37
Calculating MD5 checksum of an ATerm: atsum	37
Calculating differences between two ATerms: atdiff	37
Using the Java ATerm Library	38
Overview of the Java ATerm Library	38
Java ATerm Interfaces	39
Example Using the Java ATerms	42
Differences between C and Java Version of ATerm Library	43
Historical Notes	44
Bibliography	44

Introduction

Cut and paste operations on complex data structures are standard in most desktop software environments: one can easily clip a part of a spreadsheet and paste it into a text document. The exchange of complex data is also common in distributed applications: complex queries, transaction records, and more complex data are exchanged between different parts of a distributed application. Compilers and programming environments consist of tools such as editors, parsers, optimizers, and code generators that exchange syntax trees, intermediate code, and the like.

Annotated Terms (ATerms) provide a solution for implementation needs in the areas of compilers, interactive programming environments and distributed applications but are more widely applicable in areas like model checking and ontology definition. They have the following characteristics:

Open	Independent of any specific hardware or software platform.
Simple	The procedural interface should contain 10 rather than 100 functions.
Efficient	Operations on data structures should be fast as possible.
Concise	Inside an application the storage of data structures should be as small as possible by using compact representations and by exploiting sharing. Between applications the transmission of data structures

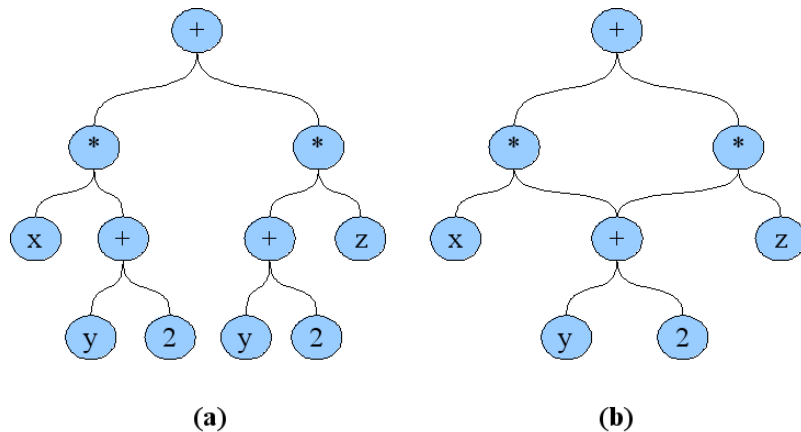
should be fast by using a compressed representation with fast encoding and decoding. Transmission should preserve any sharing of in-memory representation in the data structures.

Language-independent	Data structures can be created and manipulated in any suitable programming language.
Annotations	Applications can transparently extend the main data structures with annotations of their own to represent non-structural information.

Typically, we want to exchange and process tree-like data structures such as parse trees, abstract syntax trees, parse tables, generated code, and formatted source texts. The applications involved include parsers, type checkers, compilers, formatters, syntax-directed editors, and user-interfaces written in a variety of languages. Typically, a parser may add annotations to nodes in the tree describing the coordinates of their corresponding source text and a formatter may add font or color information to be used by an editor when displaying the textual representation of the tree.

The ATerm data type has been designed to represent such tree-like data structures and it is therefore very natural to use ATerms both for the internal representation of data inside an application and for the exchange of information between applications. Besides function applications that are needed to represent the basic tree structure, a small number of other primitives are provided to make the ATerm data type more generally applicable. These include integer constants, real number constants, binary large data objects ("blobs"), lists of ATerms, and place holders to represent typed gaps in ATerms. Using the comprehensive set of primitives and operations on ATerms, it is possible to perform operations on an ATerm received from another application without first converting it to an application-specific representation.

Figure 1.1. Maximal subterm sharing for $x*(y+2) + (y+2)*z$. (a) Tree representation. (b) Maximal subterm sharing



One particular aspect of ATerms makes them unique and should be mentioned here explicitly: ATerms are based on *maximal subterm sharing*. This is illustrated in Figure 1.1, "Maximal subterm sharing for $x*(y+2) + (y+2)*z$. (a) Tree representation. (b) Maximal subterm sharing" for the expression $x*(y+2) + (y+2)*z$. In (a) the ordinary tree representation is shown and in (b) the common subexpression $(y+2)$ is shared thus turning the tree into a Directed Acyclic Graph (DAG).

Maximal subterm sharing is a strategy to achieve "conciseness" as mentioned in the characteristics of ATerms above and is a simple and effective way to minimize memory usage: terms are only created when they are *new*, i.e., do not exist already. If a term to be constructed already exists, that term is reused, ensuring maximal sharing. This strategy fully exploits the redundancy that is typically present in the terms to be built and leads to maximal sharing of subterms. The library functions that construct terms always return maximally shared terms as result. The sharing of terms is thus invisible to the library user. Apart from *reduced memory usage*, maximal subterm sharing has another benefit: the

equality check on terms becomes very cheap: it reduces from an operation that is linear in the number of subterms to be compared to a constant operation (only pointer equality has to be checked).

Maximal subterm sharing has, however, a price. In order to optimize sharing, ATerms are immutable and "updating" an ATerm in any way always leads to the construction of a new ATerm that may, however, share many subterms with the original ATerm. ATerm construction is also penalized by an additional lookup operation to check for existing terms.

Despite these disadvantages, there are many application areas where ATerms lead to more transparent program code and to more efficient execution. Some examples are term rewriting and model checking, see [BK07] (page 44) for these and other examples.

Where to go from here?

- Read the section called "ATerms at a glance" (page 3) to get a quick overview of the functionality of ATerms. These and the following sections are written from the C perspective.
- Read the section called "Using the C ATerm Library" (page 9) for details about initialization, memory management and file format.
- A detailed description of the C ATerm library can be found in the section called "Level One Interface of C ATerm Library" (page 13) and the section called "Level Two Interface of C ATerm Library" (page 22).
- An overview of the Java ATerm library is given in the section called "Using the Java ATerm Library" (page 38) It also summarizes the differences between the C and the Java version, see the section called "Differences between C and Java Version of ATerm Library" (page 43).
- With the section called "Historical Notes" (page 44) and the section called "Bibliography" (page 44) we complete this ATerm programming guide.

ATerms at a glance

We now describe the constructors of the ATerm data type and the operations defined on it.

The ATerm data type

The data type of ATerms (ATerm) is defined as follows:

INT	An integer constant is an ATerm.
REAL	A real constant is an ATerm.
APPL	A function application consisting of a function symbol and zero or more ATerms (arguments) is an ATerm. The number of arguments of the function is called the <i>arity</i> of the function.
LIST	A list of zero or more ATerms is an ATerm.
PLACEHOLDER	A placeholder term containing an ATerm representing the type of the placeholder is an ATerm.
BLOB	A "blob" (Binary Large data Object) containing a length indication and a byte array of arbitrary (possibly very large) binary data is an ATerm.
ANNOTATION	A list of ATerm pairs may be associated with every ATerm representing a list of (<i>label, annotation</i>) pairs.

Each of these constructs except the last one (i.e., INT, REAL, APPL, LIST, PLACEHOLDER, and BLOB) form subtypes of the data type ATerm. These subtypes are needed when determining the type of an arbitrary ATerm. Depending on the actual implementation language the type is represented as a constant (C) or a subclass (Java, C#). The last construct is the *annotation construct* which makes it possible to annotate terms with transparent information. We will now give a number of examples to show some of the features of the textual representation of ATerms.

Note

The textual representation of ATerms is readable and is useful for explanation and debugging. It is hardly ever used in large applications. See the section called “ATerm formats” (page 11) for the various ATerm formats.

- Integer and real constants are written conventionally: 1, 3.14, and -0.7E34 are all valid ATerms.
- Function applications are represented by a function name followed by an open parenthesis, a list of arguments separated by commas, and a closing parenthesis. When there are no arguments, the parentheses may be omitted. Examples are: `f(a,b)` and `"test!"(1,2.1,"Hello world!")`. These examples show that double quotes can be used to delimit function names that are not identifiers.
- Lists are represented by an opening square bracket, a number of list elements separated by commas and a closing square bracket: `[1,2,"abc"]`, `[]`, and `[f,g([1,2]),x]` are examples.
- A placeholder is represented by an opening angular bracket followed by a subterm and a closing angular bracket. Examples are `<int>`, `<[3]>`, and `<f(<int>,<real>>>`.
- Blobs do not have a concrete syntax because their human-readable form depends on the actual blob content.

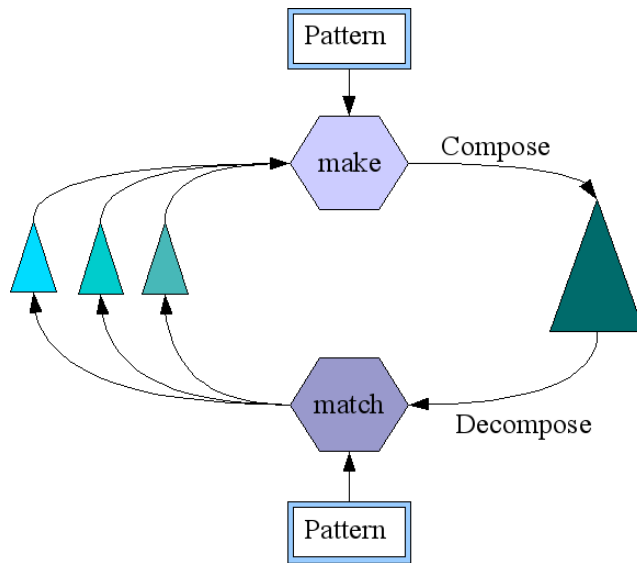
Operations on ATerms

The operations on ATerms fall into three categories: making and matching ATerms, reading and writing ATerms, and annotating ATerms. These functions provide enough functionality for most users to build simple applications with ATerms. We refer to this interface as the *level one* interface of the ATerm data type. To accommodate “power” users of ATerms we also provide a *level two* interface, which contains a more sophisticated set of data types and functions. It is typically used in generated C or Java code that calls ATerm primitives, or in efficiency-critical applications. These extensions are useful only when more control over the underlying implementation is needed or in situations where some operations that can be implemented using level one constructs can be expressed more concisely and implemented more efficiently using level two constructs. The level two interface is a strict superset of the level one interface. Observe that ATerms are a purely functional data type and that no destructive updates are possible.

Making and Matching ATerms

The simplicity of the level one interface is achieved by the *make-and-match* paradigm that is also illustrated in Figure 1.2, “The make-and-match paradigm” (page 5):

- | | |
|-------|--|
| Make | Compose a new ATerm by providing a pattern for it and filling in the holes in the pattern. |
| Match | Decompose an existing ATerm by comparing it with a pattern and decompose it according to this pattern. |

Figure 1.2. The make-and-match paradigm

Composition and decomposition of terms is *not* based on the direct manipulation of the underlying representation of terms. Instead, *term patterns* are used to guide composition and decomposition. Such term patterns play the same role as format strings in the `printf/scanf` paradigm in C. In a first approximation, a term pattern is a literal string that would be obtained by a preorder traversal of a term. For instance, the term pattern

```
"or(true, false)"
```

corresponds to a term whose root is labeled with the symbol `or`, and whose children are labeled with, respectively, `true` and `false`. In this way, term patterns can be used to construct and to match terms. Term patterns become, however, much more useful if they can be parameterized with subterms that have been computed separately. To this end, we introduce the notion of *directives* as follows:

- `<int>`: corresponds to an integer (in C: `int`);
- `<str>`: corresponds to a string (in C: `char *`);
- `<blob>` corresponds to a binary string (in C: a (length, pointer) pair represented by two values of types, respectively, `int` and `void *`);
- `<term>`: corresponds to an ATerm (in C: `ATerm`);
- `<appl>`: corresponds to one function application (in C: `char *pattern`, followed by arguments);
- `<list>`: corresponds to a list of terms (in C: `ATerm`).

The precise interpretation of these directives depends on the context in which they are used. When constructing a term, directives indicate that a subterm should be obtained from some given variable. When matching a term, directives indicate the assignment of subterms to given variables.

Patterns are just ATerms containing place holders. These place holders determine the places where ATerms must be substituted or matched. An example of a pattern is `"and(<int>, <appl>)"`. These patterns appear as string argument of both `make` and `match` and are remotely comparable to the format strings in the `printf/scanf` functions in C. The operations for making and matching ATerms are:

- `ATerm ATmake(String p, ATerm a1, ..., ATerm an)`: Create a new term by taking the string pattern `p`, parsing it as an ATerm and filling the place holders in the resulting term with values taken from `a1` through `an`. If the parse fails, a message is printed and the program is aborted.

The types of the arguments depend on the specific place holders used in the pattern p . For instance, when the placeholder `<int>` is used an integer is expected as argument and a new integer ATerm is constructed.

- `ATbool ATmatch(ATerm t, String p, ATerm *a1, ..., ATerm *an):` Match term t against pattern p , and bind subterms that match with place holders in p with the result variables a_1 through a_n . Again, the type of the result variables depends on the place holders used. If the parse of pattern p fails, a message is printed and the program is aborted. If the term itself contains place holders these may occur in the resulting substitutions. The function returns true when the match succeeds, false otherwise.

For instance, assuming the declarations

```
int n = 10;
char *fun = "pair", name = "any";
ATerm yellow = ATmake("yellow"), t;
```

the call

```
t = ATmake("exam(<appl(<term>,9)>,<int>,<str>)",
          fun, yellow, n, 10, name)
```

will construct the term t with value

```
exam(pair(yellow,9),10,10,"any")
```

Binary strings (*Binary Large Objects* or *blobs*) are used to represent arbitrary length, binary data that cannot be represented by ordinary C strings because they may contain `  null` characters. A binary string is represented by a character pointer and a length. For instance, given

```
char buf[12];
ATerm bstr;
buf[0] = 0; buf[1] = 1; buf[2] = 2;
```

the call

```
bstr = ATmake("exam(<blob>)", 3, buf);
```

will construct a term with function symbol `exam` and as single argument a binary string of length 3 consisting of the three values 0, 1, and 2.

Matching terms amounts to

- determining whether there is a match or not,
- selectively assigning matched subterms to given variables.

For instance, in the context

```
ATerm t = ATmake("exam(pair(yellow,9),10, \"any\")");
ATerm t1;
int n;
char *ex, *s;
```

the call

```
ATmatch(t, "appl(<term>,<int>,<str>)", &ex, &t1, &n, &s);
```

yields true and is equivalent to the following assignments:

```
ex = "exam";
t1 = ATmake("pair(yellow,9)");
```

```
n = 10;
s = "any";
```

As explained in full detail in the section called “Memory Management of ATerms” (page 10) memory is managed automatically by the ATerm library. As a general rule, the values for `ex`, `t1`, and `s` are pointers into the original term `t` rather than newly created values. As a result, they have a life time that is equal to that of `t`. Matching binary strings is the inverse of constructing them. Given the term `bstr` constructed at the end of the previous paragraph, its size and contents can be extracted as follows:

```
int n;
char *p;

ATmatch(bstr, "exam(<blob>)", &n, &p);
```

`ATmatch` will succeed and will assign 3 to the variable `n` and will assign a pointer to the character data in the binary string to the variable `p`. Here, again, the value of `p` is a pointer into the term `bstr` rather than a newly allocated string. Notes

- Double quotes (“”) appearing *inside* the pattern argument of both `ATmake` and `ATmatch` have to be escaped using “\”.
- The number and type of the variables whose addresses appear as arguments of `ATmatch` should correspond, otherwise disaster will strike (as usual when using C).
- Assignments are being made during matching. As a result, some assignments may be performed, even if the match as a whole fails.

Reading and Writing ATerms

For reasons of efficiency and conciseness, reading and writing can take place in several formats, see the section called “ATerm formats” (page 11) Either format (textual or binary) can be used on any linear stream, including files, sockets, pipes, etc. Here, we only give examples of the pure textual representation of ATerms.

The operations for reading and writing ATerms are:

- `ATerm ATreadFromString(String s)`: Creates a new term by parsing the string `s`. When a parse error occurs, a message is printed, and a special error value is returned.
- `ATerm ATreadFromTextFile(File f)`: Creates a new term by parsing the data from file `f`. Again, parse errors result in a message being printed and an error value being returned.
- `String ATwriteToString(ATerm t)`: Return the text representation of term `t` as a string.
- `ATbool ATwriteToTextFile(ATerm t, File f)`: Write the text representation of term `t` to file `f`. Returns true for success and false for failure.

For instance, in the context:

```
FILE *f = fopen("foo", "wb");
ATerm Trm1 = ATmake("<appl(red,<int>>)", "freq", 17);
```

the statement

```
ATwriteToTextFile(Trm1, f);
```

will write the value of `Trm1` (i.e., `freq(red, 17)`) to file “foo”.

When end of file is encountered or the term could not be read, the operation is aborted. The user can redefine this behaviour using `ATsetAbortHandler`, which allows the definition of a user-defined abort handler. See the section called “`ATsetAbortHandler`” (page 21) for further details.

The last form of output that is supported by the ATerm library is formatted output. The function

```
int ATfprintf(FILE *File, const char *Pattern, ...)
```

writes formatted output to *File*. *Pattern* is printed literally except for occurrences of directives which are replaced by the textual representation of the values appearing in *...* For instance,

```
ATfprintf(stderr, "Wrong event \"%t\" ignored\n",
           ATmake("failure(<int>)", 13));
```

will print:

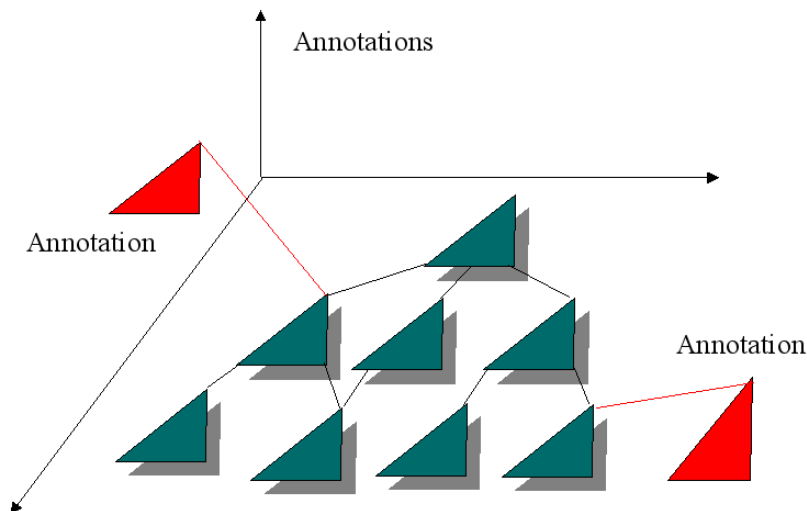
```
Wrong event "failure(13)" ignored
```

Note that `ATfprintf` uses the normal `printf` conversion specifiers extended with ATerm-specific specifiers. The most frequently used specifier is `{\tt %t}` which stands for an ATerm argument whose textual representation is to be inserted in the output stream.

Annotating ATerms

Annotations are $(label, annotation)$ pairs that may be attached to an ATerm. Annotations can be considered to be a "third dimension" for ATerms, see the section called "Annotating ATerms" (page 8). Ordinary ATerms (bottom plane) can be extended in this dimension with arbitrary ATerms (that, indeed, may again contain annotations).

Figure 1.3. ATerm annotations



The following operations for manipulating annotations are available (recall that ATerms are a completely functional data type and that no destructive updates are possible):

- ATerm `ATsetAnnotation(ATerm t, ATerm l, ATerm a)`: Return a copy of term *t* in which the annotation labeled with *l* has been changed into *a*. If *t* does not have an annotation with the specified label, it is added.
- ATerm `ATgetAnnotation(ATerm t, ATerm l)`: Retrieve the annotation labeled with *l* from term *t*. If *t* does not have an annotation with the specified label, a special error value is returned.
- ATerm `ATremoveAnnotation(ATerm t, ATerm l)`: Return a copy of term *t* from which the annotation labeled with *l* has been removed. If *t* does not have an annotation with the specified label, it is returned unchanged.

Using the C ATerm Library

Initializing and using the ATerm library

Using the ATerm library requires the following:

- Include the header file `aterm1.h` (or `aterm2.h` if you want to use the level 2 interface). `aterm1.h` defines:

- `ATbool`: the boolean data type defined by

```
typedef enum ATbool {ATfalse=0, ATtrue} ATbool;
```

It is mainly used as the return value of library functions.

- `ATerm`: the type definition of ATerms. The ATerm library has been designed in such a way that only pointers to terms must be passed to or are returned by library functions. The primitives that are provided for constructing and decomposing terms are of such a high level that it is unnecessary to know the internal representation of terms. When necessary, you can access the internal structure of ATerms using the level 2 interface.
- Declare in your main program a local ATerm variable that will be used to determine the bottom of C's runtime stack.
- Call `ATinit` to initialize the ATerm library.
- Link the ATerm library `libATerm.a` when compiling your application. This is achieved using the `-lATerm` option of the C compiler.

A typical usage pattern is as follows:

```
#include <aterm1.h>
int main(int argc, char *argv[])
{
    ATerm bottomOfStack;           ❶
    ATinit(argc, argv, &bottomOfStack); ❷
    /* ... code that uses ATerms ... */
}
```

Notes:

- ❶ The local variable `bottomOfStack` is used to indicate the bottom of C's run-time stack and needed when initializing the ATerm library.
- ❷ Initialize the ATerm library. Observe that the program arguments are passed to `ATinit` (see below).

The command line options can be passed to an application that use the ATerm library are listed in Table 1.1, “Command line options ATerm library” (page 10).

Table 1.1. Command line options ATerm library

Option	Description
<code>-at-symboltable <i>nsymbols</i></code>	Initial size of symbol table
<code>-at-termtable <i>tableclass</i></code>	Start with term table of $2^{\text{tableclass}}$ entries
<code>-at-hashinfo</code>	Write hash table statistics to the file <code>hashing.stats</code> after execution
<code>-at-print-gc-time</code>	Print timing information about garbage collector to <code>stderr</code> after execution
<code>-at-print-gc-info</code>	print verbose information about garbage collector to <code>stderr</code> after execution
<code>-at-silent</code>	Do not print status and version information

Memory Management of ATerms

The functions in the ATerm library provide automatic memory management of terms. Terms that have been created but are no longer referenced are removed by a method called *garbage collection*. There are two categories of terms that will survive a garbage collection:

- Terms that are referenced via a local variable of a currently active procedure.
- Terms that are explicitly *protected* by the user.

Effectively, all terms referenced by local variables and all protected terms (and their subterms) are conserved and all other terms are considered as garbage and can be collected. It is guaranteed that no garbage collection takes place during the execution of an event handler, hence it is not necessary to protect temporary terms that are constructed during the execution of an event handler. However, terms that should have a longer life time must be protected in order to survive. In order to protect terms from being collected, the function

```
void ATprotect(ATerm *TmPtr)
```

can be used that has as single argument *a pointer to a variable with an ATerm as value*. The protection can be undone by the function

```
void ATunprotect(ATerm *TmPtr)
```

The interplay between garbage collection and program variables is subtle. The following points are therefore worth mentioning:

- Functions that return a term as value (e.g., `ATBreadTermFromFile`) do not explicitly protect it. However, since the result will be referenced via local variable it will be safe for the garbage collector.
- The function `ATmake` uses strings and terms and includes them into a new term *T*. The implications for memory management are:
 - All string arguments (using `<str>`, `<blob>` or `<appl>`) are copied before they are included into *T*. They can thus safely be deallocated (e.g., using `free`) by the C program in case they were globally allocated.
 - All term arguments (using `<term>`) are included into *T* by means of a pointer. They thus become reachable from *T* and their life time becomes at least as large as that of *T*; it is not required to explicitly protect them unless the user decides otherwise.
- The function `ATmatch` assigns strings and terms to program variables by extracting them from an existing term *T*. The general rule here is that extracted values have a life time that is equal to that of *T*. The implications for memory management are:

- All string values (obtained using `<str>`, `<blob>` or `<appl>`) should be copied if they are used outside the scope of the function in which they were created.
- All term values (obtained using `<term>`) should be explicitly protected if they should survive *T*.

ATerm formats

ATerms can be represented in four formats:

- ASCII text (the textual representation discussed earlier) This format is human-readable, space-inefficient, and any sharing of the in-memory representation of terms is lost.
- Textual ATerm Format (TAF), a textual format that preserves maximal subterm sharing.
- Binary ATerm Format (BAF) is a portable, machine-readable, very compact format and preserves all in-memory sharing.
- Streamable ATerm Format (SAF) allows the streaming of ATerms between applications (using a fixed buffer size), also preserves sharing and optimizes the balance between speed, memory usage and compression.

We now briefly discuss each format and conclude with a decision table when to use which format.

ASCII ATerm Format (ASCII)

The simplest format available for ATerms is plain ASCII text: the ATerm is read and written in prefix form. Its main advantage are simplicity and readability. The main disadvantage is that maximal subterm sharing is lost and that size may become huge. The ASCII ATerm Format is mainly used for debugging purposes.

The main functions are:

- `ATerm ATreadFromString(String s)`: Creates a new term by parsing the string *s*. When a parse error occurs, a message is printed, and a special error value is returned.
- `ATerm ATreadFromTextFile(File f)`: Creates a new term by parsing the data from file *f*. Again, parse errors result in a message being printed and an error value being returned.
- `String ATwriteToString(ATerm t)`: Return the text representation of term *t* as a string.
- `ATbool ATwriteToTextFile(ATerm t, File f)`: Write the text representation of term *t* to file *f*. Returns true for success and false for failure.

Textual ATerm Format (TAF)

There is also a textual ATerm format which supports maximal sharing but uses a much less complex algorithm than the one used to encode and decode BAF files. This results in files that are somewhat larger than their BAF counterparts, but are often (if the terms contain redundancy) significantly smaller than their unshared form. TAF files always start with a '!' character to distinguish them from other ATerm formats. The format uses abbreviations to refer to previously written terms. An abbreviation consists of a hash character ('#') followed by a number in encoded using the Base64 Alphabet (see RFC2045). Each term whose unparsed representation would take up more bytes than the textual representation of the next available abbreviation is assigned such an abbreviation it has been written. Subsequent occurrences of this term are then written by emitting the abbreviation instead of the term itself. For example the term `f(test, test)` is represented as `!f(test, #A)` in TAF, whereas `f(a, a)` is represented as `!f(a, a)` because `test` is longer than its abbreviation `#A`, but `a` is not.

The main functions are:

- `ATerm ATreadFromSharedTextFile(FILE *f)`: Reads the TAF representation of term *t* from file *f*.

- `long ATwriteToSharedTextFile(ATerm t, FILE *f)`: Write the TAF representation of term t to file f .

Binary ATerm Format (BAF)

The ATerm library is also equipped to store and restore ATerms in a compact, portable binary representation. This representation is called BAF which stands for "Binary ATerm Format". This format can be used to write a binary version of an ATerm to file, which can later be restored in a much more efficient way than would be possible had the ATerm's textual counterpart been used. This is due to the fact that textual representations have to be (re-)parsed each time they are read from file, whereas BAF directly describes how to rebuild the internal representation of an ATerm, thus skipping the parsing phase. Moreover, the maximal sharing of ATerms is exploited when writing BAF-representations, making them take up much less space than their textual representations would have needed. Users of the ATerm library are encouraged to use BAF representations when saving ATerms to file. BAF was designed to be platform independent, which facilitates the exchange of ATerms. The ATerm library comes with a utility that is able to convert an ATerm's textual representation into its BAF counterpart and vice versa, see the section called "ATerm-conversion: **baffle**" (page 36). This conversion makes it possible to always work with BAF representations, while still being able to look at the textual representation any time an error is suspected. It also allows conversion of textual ATerms written by programs unable to write BAF which is especially convenient when these ATerms are bulky. Although the ATerm library does not impose any constraints on the names of ATerm-files, users are encouraged to use the extension `.baf` for BAF files. This will avoid confusion between textual representations and binary ones. Textual representations could use the extension `.term`. BAF files always start with a NULL character followed by `0xBAF` (hex) to distinguish them from other ATerm formats.

The available functions are:

- `ATerm ATreadFromBinaryFile(File f)`: Creates a new term by reading a BAF representation from file f .
- `ATbool ATwriteToBinaryFile(ATerm t, File f)`: Write a BAF representation of term t to file f . Returns true for success, and false for failure.

(Semi-) Streamable ATerm Format (SAF)

The (Semi-) Streamable ATerm Format is a recent addition of the library. It is designed for usage in high-performance applications and was initially developed for exchanging ATerms across network connections in a portable way. It attempts to find a balance between the following characteristics:

- Encoding / decoding speed.
- Streaming functionality.
- Compression rate.
- Memory usage.

As the name suggests, this format enables the transmission of ATerms in a semi streamlike fashion. This is achieved by reading and writing the serial representation of an ATerm in blocks of a variable size; allowing the encoding and decoding process to be suspended at any point in time.

The available functions are:

- `ATerm ATreadFromSAFFile(File f)`: Creates a new term by reading a SAF representation from file f .
- `ATbool ATwriteToSAFFile(ATerm t, File f)`: Write a SAF representation of term t to file f . Returns true for success, and false for failure.

What is the best format?

Given these four formats it is not so easy to choose the right one. Their properties are summarized in Table 1.2, “Properties of ATerm file formats” (page 13) For high performance applications BAF and SAF are the most likely candidates. BAF achieves a slightly better compression rates, but SAF is able to encode and decode ATerms much faster. Another factor to consider is that there only is a BAF implementation for C.

Tip

Since SAF is available for C and Java and is very efficient, it is a safe choice for most applications.

Warning

Refer to SAF implementation document for measurements.

Table 1.2. Properties of ATerm file formats

Property	ASCII	TAF	BAF	SAF
Readability	++	++	--	--
Efficiency writing	--	+	++	++
Efficiency reading	--	++	+	++
Compression factor	--	++	++	+
Memory usage	-	++	++	+
C implementation	+	+	+	+
Java implementation	+	+	-	+

Level One Interface of C ATerm Library

All types and functions that are defined in the level one interface are declared in `aterm1.h`. the section called “Level One Types” (page 13) reveals the types of ATerms that are used in the ATerm library, as well as the extension to the standard C-types introduced in the level one interface. To avoid confusion between BAF and the ATerm type `AT_BLOB`, the section called “A note on ‘blobs’ and BAF” (page 14) is dedicated to explaining the difference between these two notions. Finally, the section called “Level One Functionality” (page 14) describes all the functions that are available in the level one interface.

Level One Types

The following C-defines are used to represent the different ATerm types:

<code>AT_INT</code>	An ATerm of type: integer.
<code>AT_REAL</code>	An ATerm of type: real.
<code>AT_APPL</code>	An ATerm of type: function application.
<code>AT_LIST</code>	An ATerm of type: list.
<code>AT_PLACEHOLDER</code>	An ATerm of type: placeholder
<code>AT_BLOB</code>	An ATerm of type: binary large object.

The following C-types are defined in the level one interface:

`ATbool` A boolean value, either `ATtrue` or `ATfalse`.

`ATerm` An annotated term.

A note on `blobs' and BAF

Although the word *binary* is used in the abbreviations of both ``blob" and BAF, these are two very different notions. A *blob* represents an ATerm that holds binary data, with no specific meaning to the ATerm library. This notion can be used as a means of escape in case you find that you need a type of ATerm that is not on the list above. The notion of BAF is explained in the section called “Binary ATerm Format (BAF)” (page 12) and refers to a specific format used for reading and writing ATerms. Thus an ATerm of type `AT_BLOB` can be saved in BAF. It could also be written in its textual representation, although this does not guarantee that the blob will be readable, after all it represents binary data.

Level One Functionality

In this section, all functions are summarized. To obtain access to the level one interface, your application should contain `#include <aterm1.h>`.

Level one: initialization

`ATinit`

```
void ATinit(int argc, char *argv[], ATerm *bottomOfStack)
```

Initialize the ATerm library.

See the section called “Initializing and using the ATerm library” (page 9).

Level one: making and matching

`ATmake`

```
ATerm ATmake(const char *pattern, ...)
```

Create an ATerm from a string pattern and a variable number of arguments. Creates an ATerm given a pattern and corresponding values. Table 1.3, “Argument types for ATmake” (page 14) shows which patterns can be used, and which type of arguments should be passed if such a pattern is used.

Table 1.3. Argument types for ATmake

Type	Pattern	Argument
Integer	<int>	int value
Real	<real>	double value
Application	<appl>	char *pattern, arguments
String	<str>	char *pattern, arguments
List	<list>	ATerm value
Term	<term>	ATerm value
Blob	<blob>	int length, void *data
Placeholder	<placeholder>	char *type value

Types `<appl>` and `<str>` should contain a pattern consisting of the function symbol to be used and the types of the arguments. This pattern must be followed by exactly the number of arguments that are used in the pattern. The types of the arguments must match the respective types used in the pattern. Both `<appl>` and `<str>` create function applications. The difference is that `<appl>` creates one

with an *unquoted* function symbol, whereas `<str>` yields a *quoted* version. Here are some examples of ATmake:

```
#include <aterm2.h>

int    ival = 42;
char   *sval = "example";
char   *blob = "12345678";
double rval = 3.14;
char   *func = "f";

void foo()
{
    ATerm term[4];
    ATerm list[3];
    ATerm appl[3];

    term[0] = ATmake("<int>" , ival);           ❶
    term[1] = ATmake("<str>" , func);          ❷
    term[2] = ATmake("<real>", rval);          ❸
    term[3] = ATmake("<blob>", 8, blob);       ❹

    list[0] = ATmake("[]");
    list[1] = ATmake("[1,<int>,<real>]", ival, rval);
    list[2] = ATmake("[<int>,<list>]", ival+1, list[1]);

    appl[0] = ATmake("<appl>", func);
    appl[1] = ATmake("<appl(<int>>)", func, ival);
    appl[2] = ATmake("<appl(<int>, <term>, <list>>)",
                    func, 42, term[3], list[2]);

    ATprintf("appl[2] = %t\n", appl[2]);
}

int main(int argc, char *argv[])
{
    ATerm bottomOfStack;

    ATinit(argc, argv, &bottomOfStack);
    foo();
    return 0;
}
```

Notes:

- ❶ Integer value 42.
- ❷ Quoted application of "f", no arguments.
- ❸ Real value 3.14.
- ❹ Blob of size 8, and data 12345678.

ATmakeTerm

```
ATerm ATmakeTerm(ATerm pattern, ...)
```

Create an ATerm from an ATerm pattern and a variable number of arguments.

Note that *pattern* is here declared as ATerm and not as a string as in ATmake. It does not have to be parsed as in the case of ATmake and therefore ATmakeTerm is more efficient.

See the section called “ATmake” (page 14).

ATvmake

```
ATerm ATvmake(const char *pattern, va_list args)
```

Create an ATerm from a string pattern and a list of arguments} See the section called “ATmake” (page 14).

ATvmakeTerm

```
ATerm ATvmakeTerm(ATerm pattern, va_list args)
```

Create an ATerm from an ATerm pattern and a list of arguments.

See the section called “ATmake” (page 14).

ATmatch

```
ATbool ATmatch(ATerm t, const char *pattern, ...)
```

Match an ATerm against a string pattern.

Matches an ATerm against a pattern, attempting to fill the `holes'. If the ATerm matches the pattern, ATtrue is returned and the variables will be filled according to the pattern, otherwise ATfalse is returned. The <list> pattern can be used to match the tail of a list as well as a variable number of arguments in a function application. Thus the first few arguments may be matched explicitly while the tail of the arguments is directed to a list.

Here are a few examples of ATmatch:

```
#include <aterm2.h>

void foo()
{
    ATbool result;
    ATerm list;
    double rval;
    int ival;

    /* Sets result to ATtrue and ival to 16. */
    result = ATmatch(ATmake("f(16)"), "f(<int>)", &ival);

    /* Sets result to ATtrue and rval to 3.14. */
    result = ATmatch(ATmake("3.14"), "<real>", &rval);

    /* Sets result to ATfalse because f(g) != g(f) */
    result = ATmatch(ATmake("f(g)"), "g(f)");

    /* fills ival with 1 and list with [2,3] */
    result = ATmatch(ATmake("[1,2,3]"),
                    "[<int>,<list>]", &ival, &list);
}

int main(int argc, char *argv[])
{
    ATerm bottomOfStack;
```

```
    ATinit(argc, argv, &bottomOfStack);
    foo();
    return 0;
}
```

ATmatchTerm

```
ATbool ATmatch(ATerm t, ATerm pattern, ...)
```

Match an ATerm *t* against a term pattern *pattern*. Since the pattern is pre-constructed and needs not be parsed, this is a more efficient variant of `ATmatch`.

Level one: reading

ATreadFromFile

```
ATerm ATreadFromFile(FILE *file)
```

Read an ATerm from binary or text file.

This function reads an ATerm from a file. A test is performed to see if the file is in plain text, BAF, TAF, or SAF format.

ATreadFromNamedFile

```
ATerm ATreadFromNamedFile(char *filename)
```

Read an ATerm from named binary or text file.

This function reads an ATerm file *filename*. A test is performed to see if the file is in plain text, TAF, BAF, or SAF format. "-" is standard input's filename.

ATreadFromString

```
ATerm ATreadFromString(const char *string)
```

Read an ATerm from an ASCII text string.

This function parses a character string into an ATerm.

ATreadFromTextFile

```
ATerm ATreadFromTextFile(FILE *file)
```

Read an ATerm from an ASCII text file.

This function reads a text file and parses the contents into an ATerm.

ATreadFromSharedString

```
ATerm ATreadFromSharedString(const char *string, int size)
```

Read a ATerm from a string in TAF format.

This function decodes a TAF-encoded character string into an ATerm.

ATreadFromSharedTextFile

```
ATerm ATreadFromBinaryFile(FILE *file)
```

Read an ATerm from a shared text (TAF) file.

This function reads a shared text file and builds an ATerm.

ATreadFromBinaryString

```
ATerm ATreadFromBinaryString(const unsigned char *string, int size)
```

Read a ATerm from a string in BAF format.

This function decodes a BAF-encoded character string into an ATerm.

ATreadFromBinaryFile

```
ATerm ATreadFromBinaryFile(FILE *file)
```

Read an ATerm from a binary (BAF) file.

This function reads a BAF file and builds an ATerm.

ATreadFromSAFFile

```
ATerm ATreadFromSAFFile(FILE *file)
```

Read an ATerm from a streaming (SAF) file.

This function reads a SAF file and builds an ATerm.

Level one: term handling

ATgetType

```
int ATgetType(ATerm term)
```

Return the type of *term*.

A macro that returns the type of an ATerm. Result is one of `AT_APPL`, `AT_INT`, `AT_REAL`, `AT_LIST`, `AT_PLACEHOLDER`, or `AT_BLOB`.

ATisEqual

```
ATBool ATisEqual(ATerm t1, ATerm t2)
```

A macro that tests equality of ATerms *t1* and *t2*.

As ATerms are created using *maximal sharing* (see Section~\ref{sharing}), testing equality is performed in constant time by comparing the addresses of *t1* and *t2*. Note however that `ATisEqual` only returns `ATtrue` when *t1* and *t2* are completely equal, inclusive any annotations they might have!

Level one: writing

ATwriteToFile

```
ATBool ATwriteToFile(ATerm t, FILE *f)
```

Writes term *t* to file *f* in textual format.

This function writes ATerm *t* to the file *f* in textual format. This term can later be read again by `ATreadFromTextFile`.

ATwriteToNamedTextFile

```
ATbool ATwriteToNamedTextFile(ATerm t, char *filename)
```

Writes term *t* to file named *filename* in textual format.

This function writes ATerm *t* in textual representation to file *filename*. "-" is standard output's filename.

ATwriteToString

```
char *ATwriteToString(ATerm t)
```

Writes term *t* to a string.

Writes term *t* to an internal string buffer. The start of this buffer is returned. Note that the contents of this buffer are volatile and may be overwritten by any call to the ATerm library.

ATwriteToSharedTextFile

```
long ATwriteToSharedTextFile(ATerm t, FILE *f)
```

Writes term *t* to file *f* in shared Textual ATerm Format (TAF).

This function writes ATerm *t* to the file *f* in TAF format, and returns the number of characters written. This term can later be read again by `ATreadFromSharedTextFile`.

ATwriteToSharedString

```
char *ATwriteToSharedString(ATerm t, int *len)
```

Writes term *t* to a shared text string in TAF.

Writes term *t* to an internal string buffer in TAF. The start of this buffer is returned, and the length of the resulting string is stored in *len*. Note that the contents of this buffer are volatile and may be overwritten by any call to the ATerm library.

ATwriteToBinaryFile

```
ATbool ATwriteToBinaryFile(ATerm t, FILE *f)
```

Writes term *t* to file *f* in Binary ATerm Format (BAF).

This function writes ATerm *t* to the file *f* in BAF. This term can later be read again by `ATreadFromBinaryFile`.

ATwriteToNamedBinaryFile

```
ATbool ATwriteToNamedBinaryFile(ATerm t, char *filename)
```

Writes term *t* to file named *filename* in Binary ATerm Format (BAF).

This function writes ATerm *t* in binary representation to file *filename*. "-" is standard output's filename.

ATwriteToBinaryString

```
char *ATwriteToBinaryString(ATerm t, int *len)
```

Writes term *t* to a shared text string in BAF.

Writes term *t* to an internal string buffer in BAF. The start of this buffer is returned, and the length of the resulting string is stored in *len*. Note that the contents of this buffer are volatile and may be overwritten by any call to the ATerm library.

ATwriteToSAFFile

```
ATbool ATwriteToSAFFile(ATerm t, char *filename)
```

Writes term *t* to file named *filename* in Streaming ATerm Format (SAF).

This function writes ATerm *t* in streaming representation to file *filename*. "-" is standard output's filename.

Level one: Formatted output

ATprintf

```
int ATprintf(const char *format, ...)
```

ATerm version of printf.

See ATvfprintf.

ATfprintf

```
int ATfprintf(FILE *stream, const char *format, ...)
```

ATerm version of fprintf.

See ATvfprintf.

ATvfprintf

```
int ATvfprintf(FILE *stream, const char *format, va_list args)
```

ATerm version of vfprintf.

The functions ATprintf, ATfprintf and ATvfprintf are used for formatted output to file. The conversion specifiers *c*, *d*, *i*, *o*, *u*, *x*, *X*, *e*, *E*, *f*, *g*, *G*, *p*, *s* behave as can be expected from *fprintf*. In addition the conversion specifiers *a*, *h*, *l*, *n* and *t* are supported as summarized in Table 1.4, "Specifiers for print conversion" (page 20)

Table 1.4. Specifiers for print conversion

Conversion specifier	Action
a	Print the symbol of an ATerm-application
h	Print the MD5 checksum of an ATerm
l	Print an ATerm-list
n	Print information about an ATerm node
t	Print an ATerm

Level one: annotations

ATsetAnnotation

```
ATerm ATsetAnnotation(ATerm t, ATerm label, ATerm anno)
```

Annotate a term with a labeled annotation.

Creates a version of *t* that is annotated with annotation *anno* which is labeled by *label*.

ATgetAnnotation

```
ATerm ATgetAnnotation(ATerm t, ATerm label)
```

Retrieves annotation of *t* with label *label*.

This function can be used to retrieve a specific annotation of a term. If *t* has no annotations, or no annotation labeled with *label* exists, NULL is returned. Otherwise the annotation is returned.

ATremoveAnnotation

```
ATerm ATremoveAnnotation(ATerm t, ATerm label)
```

Remove a specific annotation from a term.

This function returns a version of *t* which has its annotation with label *label* removed. If *t* has no annotations, or no annotation labeled with *label* exists, *t* itself is returned.

Level one: handling warnings, errors and aborts

ATsetWarningHandler

```
void ATsetWarningHandler(void (*handler)(const char *format, va_list args))
```

Specify a warning handler for the ATerm library.

Sets a warning handler for the ATerm library. This handler will be called when an error message is issued via `ATwarning`.

ATwarning

```
void ATwarning(const char *format, ...)
```

Issue a warning message.

If an error handler has been installed through a call to `ATsetWarningHandler`, this handler will be called. Otherwise `ATwarning` uses `ATvfprintf` to print a formatted message to `stderr` and returns.

ATsetErrorHandler

```
void ATsetErrorHandler(void (*handler)(const char *format, va_list args))
```

Specify an error handler for the ATerm library.

Sets an error handler for the ATerm library. This handler will be called when an error message is issued via `ATerror`.

ATerror

```
void ATerror(const char *format, ...)
```

Issue an error message and exit the ATerm library.

If an error handler has been installed through a call to `ATsetErrorHandler`, this handler will be called. Otherwise `ATerror` uses `ATvfprintf` to print a formatted message to `stderr` and exits with errorcode 1.

ATsetAbortHandler

```
void ATsetAbortHandler(void (*handler)(const char *format, va_list args))
```

Specify an abort handler for the ATerm library.

Sets an abort handler for the ATerm library. This handler will be called when an error message is issued via `ATabort`.

ATabort

```
void ATabort(const char *format, ...)
```

Issue a error message and abort the ATerm library.

If an abort handler has been installed through a call to `ATsetAbortHandler`, this handler will be called. Otherwise `ATabort` uses `ATvfprintf` to print a formatted message to `stderr` and calls `abort`.

Level one: memory management

ATprotect

```
void ATprotect(ATerm *atp)
```

Protect an ATerm.

Protects an ATerm from being freed by the garbage collector. See the section called “Memory Management of ATerms” (page 10).

ATunprotect

```
void ATunprotect(ATerm *atp)
```

Unprotect an ATerm.

Releases protection of an ATerm which has previously been protected through a call to `ATprotect`. See the section called “Memory Management of ATerms” (page 10).

ATprotectArray

```
void ATprotectArray(ATerm *start, int size)
```

Protect an array of ATerms.

Protects an entire array of *size* ATerms starting at *start*.

ATunprotectArray

```
void ATunprotectArray(ATerm *start)
```

Unprotect an array of ATerms.

Releases protection of the array of ATerms which starts at *start*.

Level Two Interface of C ATerm Library

This section explains in detail the types and functions that are defined in the level two interface of the Term library. These functions are declared in `aterm2.h`.

Level Two Types

In addition to the C-types explained in the section called “Level One Types” (page 13) the level two interface also uses the following ATerm types:

<code>ATermInt</code>	An integer value.
<code>ATermReal</code>	A real value.
<code>ATermAppl</code>	A function application.
<code>ATermList</code>	A list of ATerms.
<code>ATermPlaceholder</code>	A placeholder.
<code>ATermBlob</code>	A Binary Large Object.

In addition to these pure ATerm types, two additional types are supported:

`ATermTable` A hash table of ATerms.

`ATermIndexedSet` A set of ATerms where each element has a unique index.

Both datatypes are *containers* for ATerms and provide mutable operations on the container itself.

Note

Of course, mutations on the containers do not affect the ATerms that occur as elements in the containers!

Level Two Functionality

This section describes all functions and macros that are available in the level two interface. To obtain access to this functionality you need to `#include <aterm2.h>` instead of `<aterm1.h>` in your application.

Level two: the type `ATermInt`

The type `ATermInt` is the ATerm representation of an integer. It abides by the rules of the C-type: `int`.

`ATmakeInt`

```
ATermInt ATmakeInt(int value)
```

Build an ATerm Int from an integer *value*.

`ATgetInt`

```
int ATgetInt(ATermInt t)
```

Macro to get the integer value from the ATerm *t*.

Level two: the type `ATermReal`

The type `ATermReal` is the ATerm representation of a real. It abides by the rules of the C-type: `double`.

`ATmakeReal`

```
ATermReal ATmakeReal(double value)
```

Build an ATerm Real from a real *value*.

`ATgetReal`

```
double ATgetReal(ATermInt t)
```

Macro to get the real value from the ATerm *t*.

Level two: the type `ATermAppl`

The type `ATermAppl` denotes a function application. In order to build a function application, first its function symbol (`AFun`) must be built. This symbol holds the name of the function application, its arity (how many arguments the function has) and whether the function name is quoted. Below are some examples of function applications and the symbols needed to create them.

- `true`: a zero arity, unquoted function application that is created by:

```
sym = ATmakeAFun("true", 0, ATfalse);
```

- "true"; the same function application, but now with quoted function symbol:

```
sym = ATmakeAFun("true", 0, ATtrue);
```

- `f(0)`: an unquoted function application of arity 1:

```
sym = ATmakeAFun("f", 1, ATfalse);
```

- `"prod"(2, b, [])`: a quoted function application of arity 3:

```
sym = ATmakeAFun("prod", 3, ATtrue);
```

ATmakeAFun

```
AFun ATmakeAFun(char *name, int arity, ATbool quoted)
```

Creates a function symbol (AFun).

Creates an AFun, representing a function symbol with name *name* and arity *arity*. Quoting of the function application is defined via the *quoted* argument.

ATprotectAFun

```
void ATprotectAFun(AFun sym)
```

Just as ATerms which are not on the stack or in registers must be protected through a call to `ATprotect`, so must AFuns be protected by calling `ATprotectAFun`.

ATunprotectAFun

```
void ATunprotectAFun(AFun sym)
```

Release an AFun's protection.

ATgetName

```
char *ATgetName(AFun sym)
```

Return the name of an AFun.

ATgetArity

```
int ATgetArity(AFun sym)
```

Return the arity (number of arguments) of a function symbol (AFun).

ATisQuoted

```
ATbool ATisQuoted(AFun sym)
```

Determine if a function symbol (AFun) is quoted or not.

ATmakeAppl

```
ATermAppl ATmakeAppl(AFun sym, ...)
```

Build an application from an AFun and a variable number of arguments.

The arity is taken from the first argument *sym*, the other arguments of `ATmakeAppl` should be the arguments for the application. For arity $N = 0, 1, \dots, 6$ the corresponding `ATmakeApplN` can be used instead for greater efficiency.

ATmakeApp10

```
ATermAppl ATmakeApp10(AFun sym)
```

Make a function application with zero arguments.

ATmakeApp11

```
ATermAppl ATmakeApp11(AFun sym, ATerm a1)
```

Make a function application with one argument.

ATmakeApp12

```
ATermAppl ATmakeApp12(AFun sym, ATerm a1, a2)
```

Make a function application with two arguments.

ATmakeApp13

```
ATermAppl ATmakeApp13(AFun sym, ATerm a1, a2, a3)
```

Make a function application with three arguments.

ATmakeApp14

```
ATermAppl ATmakeApp14(AFun sym, ATerm a1, a2, a3, a4)
```

Make a function application with four arguments.

ATmakeApp15

```
ATermAppl ATmakeApp15(AFun sym, ATerm a1, a2, a3, a4, a5)
```

Make a function application with five arguments.

ATmakeApp16

```
ATermAppl ATmakeApp16(AFun sym, ATerm a1, a2, a3, a4, a5, a6)
```

Make a function application with six arguments.

ATgetAFun

```
AFun ATgetAFun(ATermAppl appl)
```

Get the function symbol (AFun) of an application.

ATgetArgument

```
ATerm ATgetArgument(ATermAppl appl, int n)
```

Get the n -th argument of an application.

ATsetArgument

```
ATermAppl ATsetArgument(ATermAppl appl, ATerm arg, int n)
```

Set the n -th argument of an application to arg .

This function returns a copy of $appl$ with argument n replaced by arg .

ATgetArguments

```
ATermList ATgetArguments(ATermAppl appl)
```

Get a list of arguments of an application.

Return the arguments of *appl* in `ATermList` format. Note: traversing the arguments of *appl* can be done more efficiently using the `ATgetArgument` macro.

ATmakeApplList

```
ATermAppl ATmakeApplList(AFun sym, ATermList args)
```

Build an application given an `AFun` and a list of arguments.

Build an application from *sym* and the argument list *args*. Note: unless the arguments are already in an `ATermList`, it is probably more efficient to use the appropriate `ATmakeApplN`.

ATmakeApplArray

```
ATermAppl ATmakeApplArray(AFun sym, ATerm args[])
```

Build an application given an `AFun` and an array of arguments.

Level two: the type ATermList

The type `ATermList` is the `ATerm` representation of linear lists.

ATmakeList

```
ATermList ATmakeList(int n, ...)
```

Create an `ATermList` of *n* elements. The elements should be passed as arguments 1, ..., *n*.

ATmakeList0

```
ATermList ATmakeList0()
```

Macro that yields the empty list `[]`.

ATmakeList1

```
ATermList ATmakeList1(ATerm a1)
```

Construct a list of one element.

ATmakeList2

```
ATermList ATmakeList2(ATerm a1, a2)
```

Construct a list of two elements.

ATmakeList3

```
ATermList ATmakeList3(ATerm a1, a2, a3)
```

Construct a list of three elements.

ATmakeList4

```
ATermList ATmakeList4(ATerm a1, a2, a3, a4)
```

Construct a list of four elements.

ATmakeList5

```
ATermList ATmakeList5(ATerm a1, a2, a3, a4, a5)
```

Construct a list of five elements.

ATmakeList6

```
ATermList ATmakeList6(ATerm a1, a2, a3, a4, a5, a6)
```

Construct a list of six elements.

ATgetLength

```
int ATgetLength(ATermList l)
```

Macro to get the length of list *l*.

ATgetFirst

```
ATerm ATgetFirst(ATermList l)
```

Macro to get the first element of list *l*.

ATgetNext

```
ATermList ATgetNext(ATermList l)
```

Macro to get the next part (the tail) of list *l*.

ATisEmpty

```
ATbool ATisEmpty(ATermList l)
```

Macro to test if list *l* is empty.

ATgetTail

```
ATermList ATgetTail(ATermList l, int start)
```

Return the sublist from *start* to the end of *l*.

ATreplaceTail

```
ATermList ATreplaceTail(ATermList l, ATermList tail, int start)
```

Replace the tail of *l* from position *start* with *tail*.

ATgetPrefix

```
ATermList ATgetPrefix(ATermList l)
```

Return all but the last element of *l*.

ATgetSlice

```
ATermList ATgetSlice(ATermList l, int start, int end)
```

Get a portion (slice) of a list.

Return the portion of *l* that lies between *start* and *end*. Thus *start* is included, *end* is not.

ATinsert

```
ATermList ATinsert(ATermList l, ATerm a)
```

Return list l with element a inserted. The behaviour of `ATinsert` is of constant complexity. That is, the behaviour of `ATinsert` does not degrade as the length of l increases.

ATinsertAt

```
ATermList ATinsertAt(ATermList l, ATerm a, int idx)
```

Return l with a inserted at position idx .

ATappend

```
ATermList ATappend(ATermList l, ATerm a)
```

Return l with a appended to it.

`ATappend` is implemented in terms of `ATinsert` by making a new list with a as the first element and then `ATinserting` all elements from l . As such, the complexity of `ATappend` is linear in the number of elements in l . When `ATappend` is needed inside a loop that traverses a list (see Example 1.1, “Parse lists, version 1” (page 28), behaviour of the loop will demonstrate quadratic complexity.

Example 1.1. Parse lists, version 1

```
/* Example of parse_list that demonstrates quadratic complexity */
ATermList parse_list1(ATermList list)
{
    ATerm    elem;
    ATermList result = ATEmpty;

    /* while list has elements */
    while (!ATisEmpty(list))
    {
        /* Get head of list */
        elem = ATgetFirst(list);

        /* If elem satisfies some predicate (not shown here)
           then APPEND it to result */
        if (some_predicate(elem) == ATtrue)
            result = ATappend(result, elem);

        /* Continue with tail of list */
        list = ATgetNext(list);
    }

    /* Return the result list */
    return result;
}
```

To avoid this behaviour, the inner loop could use `ATinsert` instead of `ATappend` to make the new list. This will cause the resulting list to be in reverse order. A single `ATreverse` must therefore be performed, but this can be done after the loop has terminated, bringing the behaviour down from quadratic to linear complexity, but at the cost of two `ATinserts` per element (one for each `ATinsert` in the loop, and an implicit one for each element through the use of `ATreverse`). An example in Example 1.2, “Parse lists, version 2” (page 29).

Example 1.2. Parse lists, version 2

```
/* Example of parse_list that demonstrates linear complexity,
 * using ATinsert instead of ATappend and reversing the list
 * outside the loop just once. */
ATermList parse_list2(ATermList list)
{
    ATerm      elem;
    ATermList  result = ATEmpty;

    /* while list has elements */
    while (!ATisEmpty(list))
    {
        /* Get head of list */
        elem = ATgetFirst(list);

        /* If elem satisfies some predicate (not shown here)
         then INSERT it to result */
        if (some_predicate(elem) == ATtrue)
            result = ATinsert(result, elem);

        /* Continue with tail of list */
        list = ATgetNext(list);
    }

    /* Return result after reversal */
    return ATreverse(result);
}
```

An even further optimisation could make use of a locally allocated buffer. While traversing the list, all elements that would normally be `ATappend`d, are now placed in this buffer. Finally, the result is obtained by starting with an empty list and `ATinsert`ing all elements from this buffer in reverse order. As the cost of allocating and freeing a local buffer is by no means marginal, this solution should probably only be applied when the loop appends more than just a few elements. This is shown in Example 1.3, “Parse lists, version 3” (page 30)

Example 1.3. Parse lists, version 3

```

/* Example of parse_list that demonstrates linear complexity,
 * but which avoids using ATinsert twice, by inlining ATreverse
 * using a local buffer. */
ATermList parse_list3(ATermList list)
{
    int      pos = 0;
    ATerm    elem;
    ATerm    *buffer = NULL;
    ATermList result = AEmpty;

    /* Allocate local buffer that can hold all elements of list */
    buffer = (ATerm *) calloc(ATgetLength(list), sizeof(ATerm));
    if (buffer == NULL) abort();

    /* while list has elements */
    while (!ATisEmpty(list))
    {
        /* Get head of list */
        elem = ATgetFirst(list);

        /* If elem satisfies some predicate (not shown here)
         * then add it to buffer at next available position */
        if (some_predicate(elem) == ATtrue)
            buffer[pos++] = elem;

        /* Continue with tail of list */
        list = ATgetNext(list);
    }

    /* Now insert all elems in buffer to result */
    for(--pos; pos >= 0; pos--)
        result = ATinsert(result, buffer[pos]);

    /* Release allocated resources */
    free(buffer);

    /* Return result */
    return result;
}

```

ATconcat

```
ATermList ATconcat(ATermList l1, ATermList l2)
```

Return the concatenation of *l1* and *l2*.

ATindexOf

```
int ATindexOf(ATermList l, ATerm a, int start)
```

Return the index of an ATerm in a list. Return the index where element *a* can be found in the list *l*. Start looking at position *start*. Returns -1 if *a* is not in the list.

ATlastIndexof

```
int ATlastIndexof(ATermList l, ATerm a, int start)
```

Return the index of an ATerm in a list (searching in reverse order).

Search backwards for element *a* in the list *l*. Start searching at position *start*. Return the index of the first occurrence of *a* encountered, or -1 when *a* is not present before *start*.

ATelementAt

```
ATerm ATelementAt(ATermList l, int idx)
```

Return a specific element of a list. Return the element at position *idx* in list *l*. Return NULL when *idx* is not in the list.

ATremoveElement

```
ATermList ATremoveElement(ATermList l, ATerm a)
```

Return the list *l* with one occurrence of element *a* removed.

ATremoveAll

```
ATermList ATremoveAll(ATermList l, ATerm a)
```

Return the list *l* with all occurrences of element *a* removed.

ATremoveElementAt

```
ATermList ATremoveElementAt(ATermList l, int idx)
```

Return the list *l* with the element at position *idx* removed.

ATreplace

```
ATermList ATreplace(ATermList l, ATerm a, int idx)
```

Return the list *l* with the element at position *idx* replaced by *a*.

ATreverse

```
ATermList ATreverse(ATermList l, ATerm a, int idx)
```

Return the list *l* with its elements in reverse order.

ATsort

```
ATermList ATsort(ATermList l, int (*compare)(ATerm t1, const ATerm t2))
```

Sort the list *l* given a comparison function *compare*. The result is a new list.

ATfilter

```
ATermList ATfilter(ATermList l, ATbool (*predicate)(ATerm t))
```

Create a new list that consists of all elements of list *l* that satisfy the predicate *predicate*.

Level two: the type ATermPlaceholder

A placeholder is a special subtype used to indicate a typed hole in an ATerm. This can be used to create a term of a specific type, even though its actual contents are not filled in.

ATmakePlaceholder

```
ATermPlaceholder ATmakePlaceholder(ATerm type)
```

Build an ATerm Placeholder of a specific type. The type is taken from the *type* parameter. See Example 1.4, “Examples of place holders” (page 32).

ATgetPlaceholder

```
ATerm ATgetPlaceholder(ATermPlaceholder ph)
```

Get the type of an ATerm Placeholder.

Example 1.4. Examples of place holders

```
#include <assert.h>
#include <aterm2.h>

/* This example demonstrates the use of an ATermPlaceholder.
 * It creates the function application "add" defined on two
 * integers without actually using a specific integer:
 * add(<int>,<int>).
 */
void demo_placeholder()
{
    Symbol          sym_int, sym_add;
    ATermAppl      app_add;
    ATermPlaceholder ph_int;

    /* Construct placeholder <int> using zero-arity function
       symbol "int" */
    sym_int = ATmakeSymbol("int", 0, ATfalse);
    ph_int = ATmakePlaceholder((ATerm)ATmakeAppl0(sym_int));

    /* Construct add(<int>,<int>) using function symbol
       "add" with 2 args */
    sym_add = ATmakeSymbol("add", 2, ATfalse);
    app_add = ATmakeAppl2(sym_add, (ATerm)ph_int, (ATerm)ph_int);

    /* Equal to constructing it using the level one interface */
    assert(ATisEqual(app_add, ATparse("add(<int>,<int>"))));

    /* Prints: Placeholder <int> is of type: int */
    ATprintf("Placeholder %t is of type: %t\n",
             ph_int, ATgetPlaceholder(ph_int));
}

int main(int argc, char *argv[])
{
    ATerm bottomOfStack;

    ATinit(argc, argv, &bottomOfStack);
    demo_placeholder();
    return 0;
}
```

Level two: the type ATermBlob

ATmakeBlob

```
ATermBlob ATmakeBlob(unsigned int size, void *data)
```

Build a Binary Large Object given *size* (in bytes) of *data*. This function can be used to create an ATerm of type blob, holding the data pointed to by *data*. No copy of this data area is made, so the user should allocate this himself. The size of a blob is limited by the maximal value of integers.

ATgetBlobData

```
void *ATgetBlobData(ATermBlob blob)
```

Macro to get the data section of a given *blob*.

ATgetBlobSize

```
int ATgetBlobSize(ATermBlob blob)
```

Macro to get the size (in bytes) of the data section of a given *blob*.

ATregisterBlobDestructor

```
void ATregisterBlobDestructor(ATbool (*destructor)(ATermBlob))
```

Register a blob-destructor function. When a blob-destructor function has been registered, it will be called whenever the garbage collector deletes an `ATermBlob`. The destructor function can then handle the deletion of the data area of the blob. At most 16 blob destructor functions can be registered in the current implementation.

ATunregisterBlobDestructor

```
void ATunregisterBlobDestructor(ATbool (*destructor)(ATermBlob))
```

Unregister a blob-destructor function that has been previously registered through a call to `ATregisterBlobDestructor`.

Level two: the type `ATermDictionary`

Dictionaries are data structures which allow looking up a certain `ATerm` given another `ATerm`. The dictionary itself is also an `ATerm` and as such is subject to the garbage collection rules of the `ATerm`. Each dictionary consists of its own list of `ATerms`. For each lookup in the dictionary, the list is traversed to see if the current element's key matches the one being looked up. A lookup in a dictionary demonstrates behaviour linear in the number of elements the dictionary contains. On average fifty percent of the number of elements in the dictionary are examined before a match is found (if the element is present at all). For a more efficient `ATerm`-to-`ATerm` mapping, see the section called “Level two: the type `ATermTable`” (page 34).

ATdictCreate

```
ATerm ATdictCreate()
```

Create a new dictionary.

ATdictGet

```
ATerm ATdictGet(ATerm dict, ATerm key)
```

Get the value belonging to a given key in a dictionary.

ATdictPut

```
ATerm ATdictPut(ATerm dict, ATerm key, ATerm value)
```

Add / update a *(key, value)*-pair in a dictionary. If *key* does not already exist in the dictionary, this function adds the *(key, value)*-pair to the dictionary. Otherwise, it updates the value associated with *key* to *value*. The modified dictionary is returned.

ATdictRemove

```
ATerm ATdictRemove(ATerm dict, ATerm key)
```

Remove a (*key*, *value*)-pair from a dictionary. If the entry was actually in the dictionary, the modified dictionary is returned. If the entry was not in the dictionary, the (unmodified) dictionary itself is returned.

Level two: the type ATermTable

The dictionaries described in the section called “Level two: the type ATermDictionary” (page 33) are in essence nothing more than linked lists, which makes them less suitable for large ATerm-to-ATerm mappings. To this end, ATerm tables were created. These are efficiently implemented using a hash table requiring approximately 16 bytes per stored entry, assuming that the hash table is filled for 50%.

ATtableCreate

```
ATermTable ATtableCreate(int initial_size, int max_load_pct)
```

Create an ATermTable given an initial size and a maximum load percentage. Whenever this percentage is exceeded (which is detected when a new entry is added using ATtablePut), the table is automatically expanded and all existing entries are rehashed into the new table. If you know in advance approximately how many items will be in the table, you may set it up in such a way that no resizing (and thus no rehashing) is necessary. For example, if you expect about 1000 items in the table, you can create it with its initial size set to 1333 and a maximum load percentage of 75%. You are not required to do this, it merely saves a runtime expansion and rehashing of the table which increases efficiency.

ATtableDestroy

```
void ATtableDestroy(ATermTable table)
```

Destroy an ATermTable. As opposed to ATermDictionaries, ATermTables are themselves *not* ATerms. This means they are *not* freed by the garbage collector when they are no longer referred to. Therefore, when the table is no longer needed, the user should release the resources allocated by the table by calling ATtableDestroy. All references the table has to ATerms will then also be removed, so that those may be freed by the garbage collector (if no other references to them exist of course).

ATtableReset

```
void ATtableReset(ATermTable table)
```

Reset an ATermTable. This function resets an ATermTable, without freeing the memory it occupies. Its effect is the same as the subsequent execution of a destroy and a create of a table, but as no memory is released and obtained from the C memory management system this function is generally cheaper. However, if subsequent tables differ very much in size, the use of ATtableDestroy and ATtableCreate may be preferred, because in such a way the sizes of the table adapt automatically to the requirements of the application.

ATtablePut

```
void ATtablePut(ATermTable table, ATerm key, ATerm value)
```

Add / update a (*key*, *value*)-pair in a table. If *key* does not already exist in the table, this function adds the (*key*, *value*)-pair to the table. Otherwise, it updates the value associated with *key* to *value*.

ATtableGet

```
ATerm ATtableGet(ATermTable table, ATerm key)
```

Get the value associated with a given *key* in a *table*.

ATtableRemove

```
void ATtableRemove(ATermTable table, ATerm key)
```

Remove the pair with the key *key* from *table*.

ATtableKeys

```
ATermList ATtableKeys(ATermTable table)
```

Get an ATermList of all the keys in a table. This function can be useful if you need to iterate over all elements in a table.

ATtableValues

```
ATermList ATtableValues(ATermTable table)
```

Get an ATermList of all the values in a table. This function can be useful if you need to iterate over all values in a table.

Level two: the type ATermIndexedSet

The data type ATermIndexedSet provides a mapping from ATerms to integers, with as aim to assign successive integers from zero upwards to each entered term. The association between a term and an integer remains fixed until the term is removed from the table. When assigning integers to newly entered elements, integers previously assigned to removed elements are used first. The range of assigned integers is thus as compact as possible. This data type can be used for various purposes. First, one can make a mapping from ATerms to elements in any arbitrary domain *D*. By entering the ATerms in an ATermIndexedSet each ATerm gets a subsequent integer assigned. These integers can be used as entries in an array to obtain the element of domain *D* that is associated with the ATerm. Another type of application is the use as a set. Suppose that a sequence of ATerms must be processed. Suppose that the sequence can contain identical ATerms, and that each unique ATerm needs to be processed only once. Each processed ATerm can then be entered in the indexed set. For each candidate ATerm to be processed one inspection of the indexed set suffices to know whether this ATerm has already been processed before. A particular instance of this kind of application is the exploration of state spaces, where each state is represented by an ATerm. The implementations of ATermIndexedSet and ATermTable are strongly related. The implementation is quite efficient both in time and space, only requiring 16 bytes for each entry in an indexed set, if the hash table, which forms its core, is half full.

ATindexedSetCreate

```
ATermIndexedSet ATindexedSetCreate(long initial_size,  
int max_load_pct)
```

Create a new ATermIndexedSet with approximately the size *initial_size*, where it guarantees that the internal hash table, will be filled up to *max_load_pct* percent. If needed, the size of the hash table is dynamically extended to hold the entries inserted into it. If extension of the hash table fails due to lack of memory, it is attempted to fill the hash table up to 100%. All elements entered into the indexed set are automatically protected. Note that for each ATindexedSetCreate an ATindexedSetDestroy must be carried out to free memory, and to allow inserted elements to be released by the automatic garbage system of the ATerm library. Carrying out a ATindexedSetReset does not free the memory, but allows inserted elements to be garbage collected.

ATindexedSetDestroy

```
void ATindexedSetDestroy(ATermIndexedSet set)
```

Releases all memory occupied by *set*.

ATindexedSetReset

```
void ATindexedSetReset(ATermIndexedSet set)
```

Clear the hash table in the *set*, but do not release the memory. Using `ATindexedSetReset` instead of `ATindexedSetDestroy` is preferable when indexed sets of approximately the same size are being used.

ATindexedSetPut

```
long ATindexedSetPut(ATermIndexedSet set, ATerm elem, ATbool *isnew)
```

Enter *elem* into the *set*. If *elem* was already in the set the previously assigned index of *elem* is returned, and *isnew* is set to false. If *elem* did not yet occur in *set* a new number is assigned and returned, and *isnew* is set to true. This number can either be the number of an element that has been removed, or, if such a number is not available, the lowest not-used number. The lowest number that is used is 0.

ATindexedSetGetIndex

```
long ATindexedSetGetIndex(ATermIndexedSet set, ATerm elem)
```

Find the index of *elem* in *set*. When *elem* is not in the set, a negative number is returned.

ATindexedSetGetElem

```
ATerm ATindexedSetGetElem(ATermIndexedSet set, long index)
```

Retrieve the element at *index* in *set*. This function must be invoked with a valid index and it returns the element assigned to this index. If it is invoked with an invalid index, effects are not predictable.

ATindexedSetRemove

```
void ATindexedSetRemove(ATermIndexedSet set, ATerm elem)
```

Remove *elem* from *set*. If a number was assigned to *elem*, it is freed to be reassigned to an element that may be put into the set at some later on.

ATindexedSetElements

```
ATermList ATindexedSetElements(ATermIndexedSet set)
```

Retrieve all elements in *set*. The resulting list is ordered from element with index 0 onwards.

Command Line Utilities

This section describes the utilities that come with the ATerm library. These utilities are automatically built when the ATerm library is compiled and installed.

ATerm-conversion: baffle

This utility can be used to convert between the different ATerm formats: TEXT, BAF, and TAF. Usage:

```
baffle [-i <input>] [-o <output> | -c] [-v] [-rb | -rt | -rs | -rS]
        [-wb | -wt | -ws | -wS]
```

The options are explained in Table 1.5, “Command line options **baffle**” (page 37).

Table 1.5. Command line options **baffle**

Option	Description
<code>-i input</code>	Read input from file <i>input</i> (default: stdin)
<code>-o output</code>	Write output to file <i>output</i> (default: stdout)
<code>-c</code>	Check validity of input-term
<code>-v</code>	Print version information
<code>-h</code>	Display help information
<code>-rb, -rt, -rs, -rS</code>	Choose between BAF, TEXT, TAF or SAF input (default: auto detected)
<code>-wb, -wt, -ws, -wS</code>	Choose between BAF, TEXT, TAF or SAF output (Default: -wb)

Some small scripts are included which can be used to connect a process producing one ATerm format to a process which expects another. These scripts just set up **baffle** with the appropriate switches and redirect `stdin` and `stdout` accordingly. These scripts are appropriately called: **trm2baf**, **baf2trm**, **trm2taf**, **taf2trm**, **baf2taf**, and **taf2baf**.

Warning

Do we want to add **trm2saf**, **saf2baf**, etc?

Calculating the size of an ATerm: **termsize**

termsize can be used to calculate three things:

- core size: the amount of memory a given ATerm needs;
- text size: the amount of memory needed to hold a textual representation of an ATerm;
- tree depth: the maximum depth of an ATerm.

Usage:

```
termsize < inputfile
```

termsize reads an ATerm from standard input (*inputfile*) and writes the results to standard output (`stdout`). The input term can be in any format (TEXT, BAF, TAF, SAF).

Calculating MD5 checksum of an ATerm: **atsum**

atsum calculates and prints the MD5 checksum of the TAF representation of an ATerm. The algorithm used is the RSA Data Security, Inc. MD5 Message-Digest Algorithm (see RFC1321). Usage:

```
atsum [inputfile]
```

Calculating differences between two ATerms: **atdiff**

atdiff compares two terms and prints a template term that covers the common parts containing placeholders of the form `<diff>` for subterms that differed, and a list of their differing subterms. Usage:

```
atdiff [<options>] file1 file2
```

The options are explained in Table 1.6, “Command line options **atdiff**” (page 38).

Table 1.6. Command line options atdiff

Option	Description
<code>--nodiffs</code>	Do not generate diffs
<code>--diffs <i>diff-file</i></code>	Write diffs to <i>diff-file</i> (default: <code>stdout</code>)
<code>--notemplate</code>	Do not generate templates
<code>--template <i>template-file</i></code>	Write templates to <i>template-file</i> (default: <code>stdout</code>)

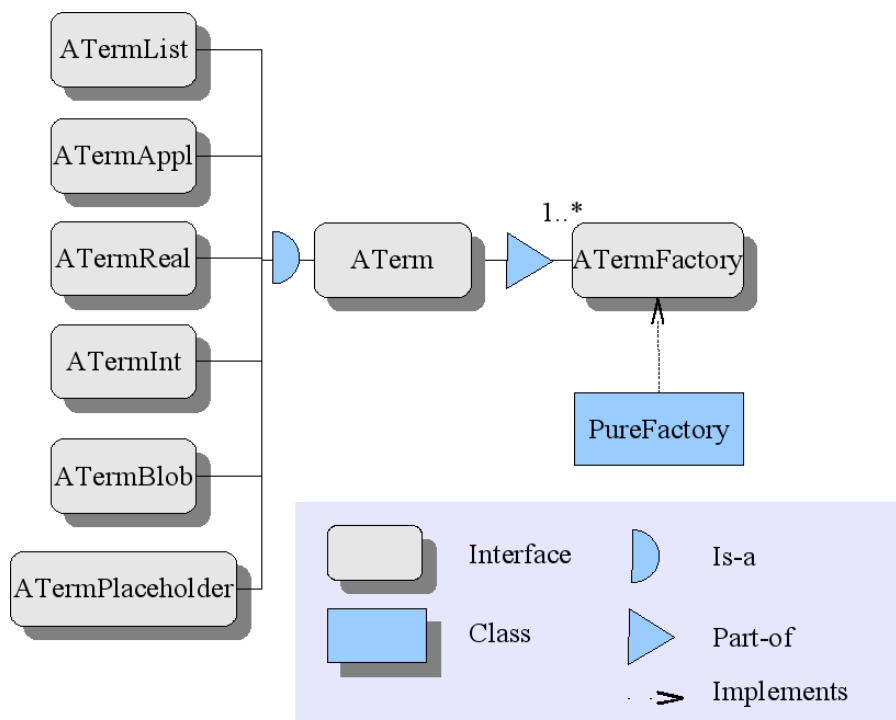
Using the Java ATerm Library

In addition to the C implementation discussed up to now, a Java implementation of the ATerm Library is also available. The interfaces of the C implementation and the Java implementation are as similar as possible. Unfortunately, constraints imposed by both languages prohibit the use of a single interface for both languages. In this section we will discuss the Java interface, and highlight the differences with the C interface where appropriate. Most differences are introduced by the fact that Java is a much more structured and higher-level language than C. For instance, Java provides built-in garbage collection, so no `ATprotect` and `ATunprotect` functions are needed in Java.

Overview of the Java ATerm Library

The interface `ATerm` defines functionality relevant for all ATerm subtypes. Each of these ATerm subtypes has its own interface, describing the additional functionality relevant for that particular subtype. An interface `ATermFactory` describes the various methods used to create new ATerm objects. It is used to implement maximal sharing.

The interface hierarchy is shown in Figure 1.4, “Interface hierarchy” (page 38).

Figure 1.4. Interface hierarchy

The ATerm library comes with a single implementation of the ATerm interfaces. For instance, the interface `ATermList` is implemented by the class called `ATermListImpl`. The `ATermFactory` is implemented by the class `PureFactory`. This implementation is a “pure” Java one, but given this

interface organization it would, in principle, be possible to build a layer of Java code on top of the C implementation using the Java Native Interface (JNI) . The advances and disadvantages of such an implementation have never been explored in detail. A complete and up-to-date description of the Java implementation of the ATerm library can be found at <http://homepages.cwi.nl/~daybuild/daily-docs/aterm-java/>.

Java ATerm Interfaces

We give here, without further ado, the methods defined by the Java ATerm Interfaces. For the meaning of individual methods we refer to both the description given for the C implementation and to <http://homepages.cwi.nl/~daybuild/daily-docs/aterm-java/>. [<http://homepages.cwi.nl/~daybuild/daily-docs/aterm-java/>]

Interface ATermInt

```
public int getInt();
```

See the section called “Level two: the type ATermInt” (page 23) for a descriptions of the corresponding C functions.

Interface ATermReal

```
public double getReal();
```

See the section called “Level two: the type ATermReal” (page 23) for a descriptions of the corresponding C functions.

Interface ATermLong

```
public long getLong();
```

The support for long integers is unique for the Java ATerm library.

Interface AFun

```
public String getName();  
public int getAriety();  
public boolean isQuoted();
```

See the section called “Level two: the type ATermAppl” (page 23) for a descriptions of the corresponding C functions.

Interface ATermAppl

```
public AFun getAFun();  
public String getName();  
public ATermList getArguments();  
public ATerm[] getArgumentArray();  
public ATerm getArgument(int i);  
public ATermAppl setArgument(ATerm arg, int i);  
public boolean isQuoted();  
public int getAriety();
```

See the section called “Level two: the type ATermAppl” (page 23) for a descriptions of the corresponding C functions.

Interface ATermBlob

```
public int getBlobSize();
```

```
public byte[] getBlobData();
```

See the section called “Level two: the type ATermBlob”(page 32)for a descriptions of the corresponding C functions.

Interface ATermFactory

```
ATerm parse(String trm);
ATerm make(String trm);
ATerm make(String pattern, List<Object> args);
ATerm make(String pattern, Object arg1);
ATerm make(String pattern, Object arg1, Object arg2);
ATerm make(String pattern, Object arg1, Object arg2, Object arg3);
ATerm make(String pattern, Object arg1, Object arg2, Object arg3,
           Object arg4);
ATerm make(String pattern, Object arg1, Object arg2, Object arg3,
           Object arg4, Object arg5);
ATerm make(String pattern, Object arg1, Object arg2, Object arg3,
           Object arg4, Object arg5, Object arg6);
ATerm make(String pattern, Object arg1, Object arg2, Object arg3,
           Object arg4, Object arg5, Object arg6, Object arg7);
ATermInt makeInt(int val);
ATermLong makeLong(long val);
ATermReal makeReal(double val);
ATermList makeList();
ATermList makeList(ATerm single);
ATermList makeList(ATerm head, ATermList tail);
ATermPlaceholder makePlaceholder(ATerm type);
ATermBlob makeBlob(byte[] data);
AFun makeAFun(String name, int arity, boolean isQuoted);
ATermAppl makeAppl(AFun fun);
ATermAppl makeAppl(AFun fun, ATerm arg);
ATermAppl makeAppl(AFun fun, ATerm arg1, ATerm arg2);
ATermAppl makeAppl(AFun fun, ATerm arg1, ATerm arg2, ATerm arg3);
ATermAppl makeAppl(AFun fun, ATerm arg1, ATerm arg2, ATerm arg3,
                  ATerm arg4);
ATermAppl makeAppl(AFun fun, ATerm arg1, ATerm arg2, ATerm arg3,
                  ATerm arg4, ATerm arg5);
ATermAppl makeAppl(AFun fun, ATerm arg1, ATerm arg2, ATerm arg3,
                  ATerm arg4, ATerm arg5, ATerm arg6);
ATermAppl makeAppl(AFun fun, ATerm[] args);
ATermAppl makeApplList(AFun fun, ATermList args);
ATerm readFromTextFile(InputStream stream)
    throws IOException;
ATerm readFromSharedTextFile(InputStream stream)
    throws IOException;
ATerm readFromFile(String file)
    throws IOException;
ATerm importTerm(ATerm term); /* from other factory */
```

See the section called “Making and Matching ATerms”(page 4)for descriptions of the make-like functions and the section called “Reading and Writing ATerms”(page 7)for the reading-writing related functions. Observe that the Java implementation does not support reading or writing from BAF files.

Interface ATermList

```
public boolean isEmpty();
```

```

public int getLength();
public ATerm getFirst();
public ATerm getLast();
public ATermList getNext();
public int indexOf(ATerm el, int start);
public int lastIndexOf(ATerm el, int start);
public ATermList concat(ATermList rhs);
public ATermList append(ATerm el);
public ATerm elementAt(int i);
public ATermList remove(ATerm el);
public ATermList removeElementAt(int i);
public ATermList removeAll(ATerm el);
public ATermList insert(ATerm el);
public ATermList insertAt(ATerm el, int i);
public ATermList getPrefix();
public ATermList getSlice(int start, int end);
public ATermList replace(ATerm el, int i);
public ATermList reverse();
public ATerm dictGet(ATerm key);
public ATermList dictPut(ATerm key, ATerm value);
public ATermList dictRemove(ATerm key);

```

See the section called “Level two: the type ATermList” (page 26) and the section called “Level two: the type ATermDictionary” (page 33) for a descriptions of the corresponding C functions. Note that ATermIndexSet (see the section called “Level two: the type ATermIndexedSet” (page 35)) is not available in the Java ATerm library.

Interface ATermPlaceholder

```

public ATerm getPlaceholder();

```

See the section called “Level two: the type ATermPlaceholder”(page 31)for a descriptions of the corresponding C functions.

Interface Identifiable

```

public int getUniqueIdentifier();

```

Interface Visitable

```

public aterm.Visitable accept(aterm.Visitor visitor)
    throws jjtraveler.VisitFailure;

```

The visitor functionality is unique for the Java ATerm library. See <http://homepages.cwi.nl/~daybuild/daily-docs/aterm-java/> for details.

Interface Visitor

```

public aterm.Visitable visitATerm(ATerm arg)
    throws VisitFailure;
public aterm.Visitable visitInt(ATermInt arg)
    throws VisitFailure;
public aterm.Visitable visitLong(ATermLong arg)
    throws VisitFailure;
public aterm.Visitable visitReal(ATermReal arg)
    throws VisitFailure;
public aterm.Visitable visitAppl(ATermAppl arg)

```

```

        throws VisitFailure;
public aterm.Visitable visitList(ATermList arg)
        throws VisitFailure;
public aterm.Visitable visitPlaceholder(ATermPlaceholder arg)
        throws VisitFailure;
public aterm.Visitable visitBlob(ATermBlob arg)
        throws VisitFailure;
public aterm.Visitable visitAFun(AFun fun)
        throws VisitFailure;

```

The visitor functionality is unique for the Java ATerm library. See <http://homepages.cwi.nl/~daybuild/daily-docs/aterm-java/> for details.

Interface ATerm

```

public static final int INT = 2;
public static final int REAL = 3;
public static final int APPL = 1;
public static final int LIST = 4;
public static final int PLACEHOLDER = 5;
public static final int BLOB = 6;
public static final int AFUN = 7;
public static final int LONG = 8;
public int getType();
public int hashCode();
public List<Object> match(String pattern);
public List<Object> match(ATerm pattern);
public boolean hasAnnotations();
public ATerm getAnnotation(ATerm label);
public ATerm setAnnotation(ATerm label, ATerm anno);
public ATerm removeAnnotation(ATerm label);
public ATermList getAnnotations();
public ATerm setAnnotations(ATermList annos);
public ATerm removeAnnotations();
public boolean isEqual(ATerm term);
public boolean equals(Object obj);
public void writeToTextFile(OutputStream stream)
        throws IOException;
public void writeToSharedTextFile(OutputStream stream)
        throws IOException;
public ATerm make(List<Object> args);
public ATermFactory getFactory();
public String toString();

```

See the section called “Making and Matching ATerms”(page 4)for descriptions of the make-like functions and the section called “Reading and Writing ATerms”(page 7)for the reading-writing related functions. Observe that LONG is only supported by the Java implementation.

Example Using the Java ATerms

To give a flavour of the manipulation of ATerms in Java, Example 1.5, “Using ATerms in Java” (page 43) shows the creation of some ATerms and reading of an ATerm from a stream.

Example 1.5. Using ATerms in Java

```

import java.io.*;
import aterm.*;

public class Basic
{
    private ATermFactory factory;

    public static final void main(String[] args) throws IOException {
        Basic basic = new Basic(args);
    }

    public Basic(String[] args) throws IOException {
        factory = new aterm.pure.PureFactory();

        ATermInt i = factory.makeInt(42);
        System.out.println("i = " + i);

        AFun fun = factory.makeAFun("foo", 2, false);
        ATermAppl foo = factory.makeAppl(fun, i, i);
        System.out.println("foo = " + foo);

        ATerm t = factory.parse("this(is(a(term(0))))");
        System.out.println("t = " + t);

        try {
            ATerm input = factory.readFromFile(System.in);
            System.out.println("You typed a valid term: " + input);
        } catch (ParseError error) {
            System.out.println("Your input was not a valid term!");
        }
    }
}

```

Differences between C and Java Version of ATerm Library

The differences between the C and Java version of the ATerm library are summarized in Table 1.7, “Differences C and Java version” (page 43).

Table 1.7. Differences C and Java version

Feature	C	Java
(Un)protecting ATerms	yes	no (automatic)
LONG	no	yes
ATermTable	yes	no
AtermIndexedSet	yes	no
BAF	yes	no
Distinction level 1/level 2 interface	yes	no

Historical Notes

The first term structure to be used for data exchange in The Meta-Environment was designed as part of the ToolBus coordination architecture and is described in [BK94] (page 44) and was implemented by Paul Klint. These "ToolBus terms" already provided the make-and-match paradigm (the section called "Making and Matching ATerms" (page 4)) for constructing and deconstructing terms. They also provided a linear string representation for the exchange of terms between components as well as automatic garbage collection.

The ATerms as discussed here are described in detail in [BJKO00] (page 44) and were implemented by Hayco de Jong and Pieter Olivier (both in C and Java). They introduced several innovations over the original design: maximal subterm sharing, annotations, a compressed binary exchange format, and a two-level Application Programming Interface (API) that enables both simple and efficient use of ATerms. The Java implementation uses SharedObjects as described in [BMV05] (page 44) and implemented by Pierre-Etienne Moreau. Erik Scheffers adapted the C ATerm Library to 64 bit architectures. Arnold Lankamp added the streamable ATerm format (SAF), performed major optimizations on both the C and the Java version and made the Java version ready for efficient execution on multi-core machines.

In order to further control the type safe access to ATerms, the API generator **apigen** has been developed. First in C, described in [JO04] (page 44) and implemented by Hayco de Jong and Pieter Olivier. Later in Java, described in [BMV05] (page 44) and implemented by Pierre-Etienne Moreau and Jurgen Vinju.

Since their inception, ATerms have been used in a wide range of application:

- Data exchange between interoperating components.
- Implementation of term rewriting languages and engines.
- Source code representation and transformation.
- Software renovation.
- Representation of web ontologies.
- Representation of state spaces that are used for model checking.
- Representation of feature diagrams as used for domain-specific engineering and software product lines.

An overview of the application of ATerms can be found in [BK07] (page 44).

Bibliography

- [BK94] J.A. Bergstra and P. Klint. *The toolbus: a component interconnection architecture*. Technical Report P9408. University of Amsterdam, Programming Research Group. 1994.
- [BJKO00] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P. Olivier. *Efficient Annotated Terms*. 259--291. *Software, Practice & Experience*. 30. 2000.
- [BMV05] M.G.J. van den Brand, P.E. Moreau, and J.J. Vinju. *A generator of efficient strongly typed abstract syntax trees in java*. 70--78. *IEE Proceedings-Software*. 152. 2. 2005.
- [BK07] M.G.J. van den Brand and P. Klint. *Aterms for manipulation and exchange of structured data: It's all about sharing*. 55--64. *Information and Software Technology*. 49. 1. 2007.
- [JO04] H.A. de Jong and P.A. Olivier. *Generation of abstract programming interfaces from syntax definitions*. 35--61. *Journal of Logic and Algebraic Programming*. 50. 4. 2004.