`adt` — library for abstract data types

Table of contents

The library adt

This library contains MuPAD implementations of abstract data types.

Every instance of these data types is realized as a MuPAD domain.

The usage of this library is completely different from the rest of the MuPAD library: An object of an `adt` data type is a domain, so that by using the methods described here, you change the object itself as a side-effect. No assignment is necessary to keep your changes. Compare this to usual MuPAD functions, where you have to always use, e.g.,

```
>> f := transform::fourier(f, x, y)
```

The data types are implemented completely within the MuPAD programming language. Keeping this in mind, the performance is excellent.

Example 1. We create an object of the abstract data type "stack" and perform the standard operations.

The stack will be initialized with the characters `"a"`, `"b"` and `"c"`:

```
>> S:= adt::Stack("a", "b", "c")
```

```
                                Stack1
```

The output `Stack1` means thet a new stack with the name `Stack1` was created and then assigned to `S`. The next stack created will get the name `Stack2` etc. The name of a stack is irrelevant. To handle the stack, it must be assigned to an identifier.

The depth (number of elements) and the top of the stack:

```
>> S::depth(), S::top()
```

```
                                3, "c"
```

Push an element, control the depth and then revert the stack. You can see that `S` is changed, e.g., when the method `"push"` is called:

```
>> S::push("d"):
   S::depth(), S::top()
```

```
                                4, "d"
```

The stack is now reverted (although this is not a standard operation for abstract stacks, it comes in handy in many uses). After that, we pop all elements until the stack is empty:

```
>> S::reverse():
```

```
>> while not S::empty() do
     print(S::pop())
   end_while;
   S::depth(), S::top()
```

"a"

"b"

"c"

"d"

0, FAIL

`adt::Queue` – abstract data type "Queue"

`adt::Queue` implements the abstract data type "Queue".

Call(s):

- `adt::Queue(queue)`

Parameters:

`queue` — an expression sequence of objects to initialize the queue

Return Value: an object of the domain `adt::Queue`

Details:

- `adt::Queue` implements the abstract data type "Queue". To create a queue, an expression sequence of any MuPAD objects can be given to initialize the queue, otherwise an empty queue is builded.
- The methods of all abstract datatypes must be called especially and will result changing the object itself as sideeffect.

- With `Q := adt::Queue()` an empty queue is builded and assigned to the variable `Q`.
- Every queue will be displayed as `Queue` followed by a number. This name is generated by `genident`.
- *All following methods changes the value of `Q` itself.* A new assignment to the variable (in this example `Q`) is not necessary, in contrast to all other MuPAD functions and data types.

- The methods `clear`, `dequeue`, `empty`, `enqueue`, `front`, `length`, `reverse` are available for handling with queues.

Method `Q::clear`: clear the queue

`Q::clear()`

- With this method the hole queue will be cleared.

Method `Q::dequeue`: get an element from the queue

`Q::dequeue()`

- This method returns the front element of the queue (the first added) with removing it from the queue.

Method `Q::empty`: is the queue empty

`Q::empty()`

- This method returns `TRUE`, if no element is in the queue, otherwise `FALSE`.

Method `Q::enqueue`: fill up the queue

`Q::enqueue(x)`

- This method puts the element `x` at the end of the queue.

Method `Q::front`: front of the queue

`Q::front()`

- This method returns the front element of the queue (the first added) *without* removing it from the queue.

Method `Q::length`: length of the queue

`Q::length()`

- This method returns the number of elements in the queue.

Method `Q::reverse`: revert the queue

`Q::reverse()`

- This method reverses the order of all elements in the queue.

Example 1. Create a new queue with strings as arguments.

```
>> Q := adt::Queue("1", "2", "3", "4")
```

```
Queue1
```

Show the length of the queue.

```
>> Q::length()
```

```
4
```

Fill up the queue with a new element. The queue will be changed by the method, no new assignment to `Q` is necessary!

```
>> Q::enqueue("5")
```

```
"5"
```

Show the front of the queue. This method does not change the queue.

```
>> Q::front(), Q::front()
```

```
"1", "1"
```

After twice getting an element of the queue, the third element is the new front of the queue, and the length is 3.

```
>> Q::dequeue(), Q::dequeue(), Q::front(), Q::length()
```

```
"1", "2", "3", 3
```

Now revert the queue. The last element will be the first element.

```
>> Q::reverse(): Q::front()
```

```
"5"
```

Enlarge the queue with "2".

```
>> Q::enqueue("2"):
   Q::empty()
```

```
FALSE
```

Finally collect all elements of the queue in the list assigned to ARGS, until the queue is empty.

```
>> ARGS := []: while not Q::empty() do ARGS := append(ARGS, Q::dequeue()) e
   ARGS
```

```
["5", "4", "3", "2"]
```

Changes:

- adt::Queue is a new function.

---

adt::Stack – **abstract data type "Stack"**

adt::Stack implements the abstract data type "Stack".

Call(s):

- adt::Stack(stack)

Parameters:

`stack` — an expression sequence of objects to initialize the stack

Return Value: an object of the domain `adt::Stack`

Details:

- `adt::Stack` implements the abstract data type "Stack". To create a stack, an expression sequence of any MuPAD objects can be given to initialize the stack, otherwise an empty stack is builded.
- The methods of all abstract datatypes must be called especially and will result changing the object itself as sideeffect.

- With `S := adt::Stack()` an empty stack is builded and assigned to the variable `S`.
- Every stack will be displayed as `Stack` followed by a number. This name is generated by `genident`.
- *All following methods changes the value of `S` itself.* A new assignment to the variable (in this example `S`) is not necessary, in contrast to all other MuPAD functions and data types.

- The methods `depth`, `empty`, `pop`, `push`, `reverse` and `top` are available for handling with stacks.

Method `S::depth`: depth of the stack

`S::depth()`

- This method returns the depth of the stack, that is the number of elements in the stack.

Method `S::empty`: is the stack empty

`S::empty()`

- This method returns `TRUE`, if no element is in the stack, otherwise `FALSE`.

Method `S::pop`: fill up the stack

`S::pop(x)`

- This method puts the element `x` at the top of the stack.

Method `S::push`: get an element from the stack

`S::push()`

- This method returns the top element of the stack (the last added) with removing it from the stack.

Method `S::reverse`: revert the stack

`S::reverse()`

- This method reverses the order of all elements in the stack.

Method `S::top`: top of the stack

`S::top()`

- This method returns the top element of the stack (the last added) *without* removing it from the stack.

Example 1. Create a new stack with strings as arguments.

```
>> S := adt::Stack("1", "2", "3", "4")
```

```
Stack1
```

Show the length of the stack.

```
>> S::depth()
```

```
4
```

Fill up the stack with a new element. The stack will be changed by the method, no new assignment to `S` is necessary!

```
>> S::push("5")
```

```
"5"
```

Show the top of the stack. This method does not change the stack.

```
>> S::top(), S::top()
```

```
"5", "5"
```

After twice getting an element of the stack, the third element is the new top of the stack, and the length is `3`.

```
>> S::pop(), S::pop(), S::top(), S::depth()
```

```
"5", "4", "3", 3
```

Now revert the stack. The last element will be the first element.

```
>> S::reverse(): S::top()
```

```
"1"
```

Fill up the stack with `"0"`.

```
>> S::push("0"):
   S::empty()
```

```
FALSE
```

Finally collect all elements of the stack in the list assigned to `ARGS`, until the stack is empty.

```
>> ARGS := []: while not S::empty() do ARGS := append(ARGS, S::pop()) end:
   ARGS
```

```
["0", "1", "2", "3"]
```

Changes:

- `adt::Stack` is a new function.

`adt::Tree` – abstract data type "Tree"

`adt::Tree` implements the abstract data type "Tree".

Call(s):

- `adt::Tree(tree)`

Parameters:

`tree` — the tree, given as a special list (see details)

Return Value: an object of the type `adt::Tree`

Related Functions: `output::tree`

Details:

- `adt::Tree` implements the abstract data type "Tree".
- A tree must be given as a special MuPAD list:
- The first object of the list is the root of the tree. All further objects are leaves or subtrees of the tree. A subtree is again a special list (as described), and any other MuPAD object will be interpreted as leaf of the tree (see example 1).
- A tree can be used to display data in tree structure using the function `output::tree` (or the method `"print"` of a tree). The nodes and leaves of the tree will be printed by MuPAD when the tree will be displayed.
- A tree can also be used as datatype to keep and handle any MuPAD data. Examples for using trees are, e.g., the library functions `matchlib::analyze` and `prog::calltree`.
- The methods of all abstract datatypes must be called especially and will result changing the object itself as sideeffect.

- `T := adt::Tree([_plus, 3, 4, [_mult, 5, 3], 1])` builds a tree and assigns it to the variable `T`.
- Every tree will be displayed as `Tree` followed by a number. This name is generated by `genident`.
- To display the content of a tree, the function `expose` or the method `"print"` of the tree itself must be used.
- *All following methods changes the value of `T` itself.* A new assignment to the variable (in this example `T`) is not necessary, in contrast to all other MuPAD functions and data types.

- The methods `nops`, `op`, `expr`, `print`, `indent`, `chars` are now available for handling with trees.

Method `T::nops`: number of operands

`T::nops()`

- This method returns the number of operands of the tree `T`. Also a subtree is an operand and will be counted as one operand.
- In this example `T` has `4` operands, the numbers `3`, `4`, `1` and the subtree `adt::Tree([_mult, 5, 3])`.

Method `T::op`: operands of a tree

`T::op(<n>)`

- This method returns the operands of the tree `T`. Also a subtree is an operand and will be returned as one operand (see `T::nops` above).
- `T::op(n)` returns the specified operands of the tree. `n` can be a number between `0` and `T::nops()` (`0` gives the root of the tree), a sequence `i..j` (to return the *i*th to *j*th operand), or a list to specify operands of subtrees (exactly as for the kernel function `op`). `T::op()` returns all operands except the `0`-th as expression sequence. See example 2.

Method `T::expr`: convert a tree to an expression

`T::expr()`

- This method returns a MuPAD expression derivated of the tree `T`. The roots of the subtrees will be interpreted as operations that will applied to the remaining operands (see example 4). In this example the corresponding expression will be `_plus(3, 4, _mult(5, 3), 1)` that is `3 + 4 + 5*3 + 1` and will be evaluated to `23`. If the roots are no operations or identifiers, the method `expr` can be result to an error.

Method `T::print`: display a tree

`T::print()`

- This method displays the tree `T` and returns the empty object `null()`.

Method `T::indent`: indent width of each operand

`T::indent(<n>)`

- `T::indent()` returns the indent width of each operand or subtree. With `T::indent(n)` the indent width will be set to the positive integer `n`.

Method `T::chars`: indent width of each operand

`T::chars(<list>)`

- `T::chars()` returns the characters that will be used to display a tree, that are `|` (vertical line), `+` (middle branch), `-` (extension of a branch), `` ` `` (last branch) and a space between the branch and the corresponding operand. With `T::chars(list)` and `list`

is a list of five characters the characters can be changed (see example 3). This list will be given as argument to `output::tree`.

Example 1. Creating a simple tree with only two leaves. To access and display a tree it must be assigned to a variable:

```
>> T := adt::Tree(["ROOT", "LEFT", "RIGHT"])

                                  Tree1
```

The tree will only be printed by its name. To display the tree, the function `expose` or the method `"print"` of the tree must be used:

```
>> T::print()

                               ROOT
                               |
                               +-- LEFT
                               |
                               '-- RIGHT

>> expose(T)

                               ROOT
                               |
                               +-- LEFT
                               |
                               '-- RIGHT
```

The next tree contains two subtrees as leaves:

```
>> T := adt::Tree(["ROOT", ["LROOT", "LLEFT", "LRIGHT"],
                           ["RROOT", "RLEFT", "RRIGHT"]]):
   T::print()

                          ROOT
                          |
                          +-- LROOT
                          |   |
                          |   +-- LLEFT
                          |   |
                          |   '-- LRIGHT
                          |
                          '-- RROOT
                              |
                              +-- RLEFT
                              |
                              '-- RRIGHT
```

Example 2. Get the operands of a tree: Also a subtree can be an operand:

```
>> T := adt::Tree(["ROOT", ["LROOT", "LLEFT", "LRIGHT"],
                           "MIDDLE",
                           ["RROOT", "RLEFT", "RRIGHT"]]):
   T::op()

                     Tree4, "MIDDLE", Tree5
```

Use `expose` to display subtrees:

```
>> map(%, expose)

                LROOT     , "MIDDLE", RROOT
                |                     |
                +-- LLEFT             +-- RLEFT
                |                     |
                `-- LRIGHT            `-- RRIGHT
```

Get all operands including the root:

```
>> T::op(0..T::nops())

                 "ROOT", Tree6, "MIDDLE", Tree7
```

Access to various operands:

```
>> T::op(0);
   T::op(2..3);
   T::op([1, 2])

                              "ROOT"

                         "MIDDLE", Tree9

                            "LRIGHT"
```

Example 3. The default characters are `["|", "+", "-", "`", " "]`:

```
>> T := adt::Tree(["ROOT", ["LROOT", "LLEFT", "LRIGHT"],
                           ["RROOT", "RLEFT", "RRIGHT"]]):
   T::print()
```

```
ROOT
|
+-- LROOT
|   |
|   +-- LLEFT
|   |
|   '-- LRIGHT
|
'-- RROOT
    |
    +-- RLEFT
    |
    '-- RRIGHT
```

The characters can be changed:

```
>> T::chars(["|", "|", "_", "|", " "]):
   T::print()
```

```
ROOT
|
|__ LROOT
|   |
|   |__ LLEFT
|   |
|   |__ LRIGHT
|
|__ RROOT
    |
    |__ RLEFT
    |
    |__ RRIGHT
```

Example 4. A tree visualizes the structure of an expression:

```
>> T:= adt::Tree([_plus, [_power, [sin, x], 2], [_power, [cos, x], 2]]):
   T::print()
```

```
_plus
|
+-- _power
|   |
|   +-- sin
|   |   |
|   |   '-- x
|   |
```

```
|   '-- 2
|
'-- _power
    |
    +-- cos
    |   |
    |   '-- x
    |
    '-- 2
```

A tree can be converted to a MuPAD expression:

```
>> T::expr(), simplify(T::expr())
```

$$\cos(x)^2 + \sin(x)^2 , \ 1$$

Changes:

- `adt::Tree` is a new function.