pm library, version 1.0

(manual)

Link to source files:
https://github.com/pellman/pm

Library and manual author:
Vladislav Podymov

August 16, 2017

Contents

1 Brief introduction

1.1 A few names

Official library abbreviated name is "pm".

At the moment pm library consists of 6 modules:

dataset, grammar, regexp, stream, type_abuse, utils.

I[1] will refer to myself as "the author" in this manual.

A "symbol" — is an object of type char.

A "string" — is an object of type std::string.

A "standard input stream" — is an object of type std::istream (or any derived class, as streams are usually being created).

1.2 A few words on pm library

The only used programming language is C++, occasionally with 11 standard syntactic features.

(Un)official expansion of the library name is "puzzling modules". Due to the meaninglessness of the official expansion is not present here, and the (un)official expansion may be considered official.

The expansion has a precise meaning: the functionality of the library

- is similar to what is provided by standard and other well-known libraries,
- but differs, somewhere significantly, and somewhere barely noticeably.

The main modules of the library by which the whole library was inspired are regexp and grammar. These modules provide a rather handy and powerful[2] parsing mechanism for strings and input streams based on regular expressions and grammars, which additionally allows to generate arbitrary data structures.

dataset module is useful mostly as the part of grammar module. It provides structures for arbitrary data collection management.

stream, type_abuse and utils modules provide classes and functions similar to standard C++ notions: a standard input stream (reversable stream in stream), smart pointers (tricky pointers in type_abuse), and other (utils). These modules are also the part of the main modules, but they might be interesting as standalone modules.

Due to dependencies between modules and their conceptionbs, the order of module descriptions is: utils, stream, type_abuse, regexp, dataset, grammar.

1.3 A few words on utils

utils contains several general purpose template functions and template classes. The author believes that these are functions and classes which hypotetically could be included in C++ standard (may be not them, but related ones), but for some reasons are not included. In details, these are:

- template functions for stl container management;[3]
- a template class Optional analogous to std::optional, which is proposed in 17 standard;[4] an object of this class (an option) contains either a value of a fixed type, or no value at all;
- a template class Numeration which is a special case of a more general popular notion: a bijection;[5] a numeration object provides a bijection between an arbitrary comparable[6] data collection and a set $\{0, 1, \ldots, k\}$.

1.4 A few words on stream

The main class of the module — a reversable stream — contains a standard input stream and an unbounded string buffer, and provides methods for unformatted input of symbols and strings (from the buffer and the standard stream) and putting these symbols and strings back (technically, to the buffer, but not the stream).

The author notes that though the standard input stream contains "almost these" methods unget and putback. The main problem of these methods is that depending on origin of the stream they may put symbols back always successfully, sometimes successfully, sometimes successfully for some symbol values, or even never put symbols back. The reversable stream class is intended to provide means for absolutely-always successfull putback.

[1] Look at the title page for my name and a contact link.

[2] At least the author thinks so.

[3] Each of the functions is basically two to five instructions of "classical" structure, but it was frustrating to write the same barely-readable classical instruction combinations in huge number of places, thus these functions.

[4] At last! A lot of programmers wanted to use this handy class in standard way, and finally the standartization machine provided it. The author does not want to wait for 17 standard to become widely used, so he implemented required functionality from scratch.

[5] Surprisingly, a bijection is not included in any standard, though the question "is there a standard bijection class?" is popular in Internet. Moreover, boost library contains a bijection template class. The author wants to keep the library independent, so he implemented a required special case of bijection from scratch.

[6] As in std::map

1.5 A few words on type_abuse

This module provide twisted versions of smart pointers — tricky pointers — which seem similar, but based on concepts hardly compatible with those of smart pointers. The main reasons to have tricky pointers is:

- to eliminate a conceptual imperfection of smart pointers which says that a pointer copy is indeed a copy (shares the same object), but a copy of an empty pointer is not a copy, but an independent empty pointer; to fix it, the author introduces basic concepts which say that the emptiness is a rightfull object — emptinesses may be equal, as well as not equal;
- to provide means for arbitrary-typed dynamically created data management;[7] this interface is extensively used in other modules, so we immediately note: a "tricky pointer to whatever data" is an object of type pm::type_abuse::DRef; due to strong static typing issues of C++ it is not really a "whatever" data, but its type and contents can be changed at any moment at runtime by a manager of a non-template[8] class.

1.6 A few words on regexp and grammar

Popular C++ means for parsing based on regular expressions and grammar combined with generation of whatever (standard regex, boost, lex/flex/bison and so on and so forth) are awful. These means are popular and widely used, and people are happy, but not the author. Some of the means are not expressive. Some of them cannot generate data in a handy way. Some of them lead to a highly unreadable or unmanageable code. Some of them restrict "parse from" and "generate to" by specific unhandy objects such as files. Some of them generate huge archaic nonintegrable unreadable pieces of code. The author can list the drawbacks forever, but still the result is the author implemented his own regular expressions and grammars,[9] which is handier than all known to the author for a certain class of parsing problems. In details, the modules provide means for parsing of strings, standard input streams, and reversable streams (module stream) based on regular expressions and grammars, combined with generation of whatever needed (precisely, of whatever data provided by module type_abuse).

2 Structure of source files

The root folder consists of:

1. An src subfolder which contains library source code and additional files required by cmake build tool.
2. A README file which redirects the reader to this manual and its initial russian version.
3. A LICENSE file which provides full text of GNU GPLv3 license under which the library is being distributed.
4. A man_ru.pdf file, which is an initial russian version of the manual.
5. A man_en.pdf file — this manual.

src subfolder consists of:

1. Subfolders dataset, grammar, regexp, stream, type_abuse and utils, in which source code of corresponding modules is located, as well as auxiliary files CMakeLists.txt required by cmake tool.
2. A file CMakeLists.txt which is the main build file for cmake tool.
3. An empty file dummy.cpp required by cmake tool for technical reasons.

3 License

The library is distributed under the terms of GNU GPLv3 license (or any later version, as written in the license). Full text of the license can be found in the LICENSE file in the root folder.

4 Requirements

pm library is designed to be highly independent. It can be built with any C++ compiler supporting the standard 11. Full support of the standard is not required. For instance, gcc 4.8.5 does not fully support the standard, and is sufficient to build the library. Earlier versions of gcc and other compilers without full support of the standard are not guaranteed to build the library, but it is not forbidden to try. If it builds, it works.

The standard syntactic features of the language are listed in the main build file in terms of cmake commands.

The rest of the section may be omitted in case the library is being built in standard ways in any standard environment.

[7] The brains of C++ continuously argue on the subject of the perfect bounds for typing restrictions, and is it good to have strong static typing and forbid to manage objects which at a given moment can have an error-prone "whatever type". Still the author had the needs to use a "whatever type", and implemented something to manage it.

[8] It is important: templates are pure evil!

[9] With blackjack and hookers.

Module ... is required by module ...

Module `dataset` is required by module `grammar`.
Module `regexp` is required by module `grammar`.
Module `stream` is required by modules `grammar`, `regexp`.
Module `type_abuse` is required by modules `dataset`, `grammar`, `regexp`.
Module `utils` is required by modules `dataset`, `grammar`, `regexp`.

Module ... depends on module ...

Module `dataset` depends on modules `type_abuse`, `utils`.
Module `grammar` depends on modules `dataset`, `regexp`, `stream`, `type_abuse`, `utils`.
Module `regexp` depends on modules `stream`, `type_abuse`, `utils`.

Standard header ... is used in module ...

`<cstddef>` is used in modules `dataset`, `grammar`, `regexp`, `stream`, `utils`.
`<functional>` is used in module `type_abuse`.
`<initializer_list>` is used in module `dataset`.
`<istream>` is used in module `stream`.
`<list>` is used in modules `dataset`, `regexp`.
`<map>` is used in modules `grammar`, `utils`.
`<memory>` is used in module `type_abuse`.
`<set>` is used in module `regexp`.
`<sstream>` is used in module `regexp`.
`<stack>` is used in module `stream`.
`<string>` is used in modules `dataset`, `grammar`, `regexp`, `stream`.
`<utility>` is used in modules `dataset`, `grammar`, `regexp`, `type_abuse`, `utils`.
`<vector>` is used in modules `dataset`, `grammar`, `utils`.

Module ... uses standard headers ...

Module `dataset` uses standard headers `<cstddef>`, `<initializer_list>`, `<list>`, `<string>`, `<utility>`, `<vector>`.
Module `grammar` uses standard headers `<cstddef>`, `<map>`, `<string>`, `<vector>`.
Module `regexp` uses standard headers `<cstddef>`, `<list>`, `<set>`, `<sstream>`, `<string>`, `<utility>`.
Module `stream` uses standard headers `<cstddef>`, `<istream>`, `<stack>`, `<string>`.
Module `type_abuse` uses standard headers `<functional>`, `<memory>`, `<utility>`.
Module `utils` uses standard headers `<cstddef>`, `<map>`, `<utility>`, `<vector>`.

5 Build

Briefly on how to build the library:

1. Take all the source files and feed them to a `C++` compiler in a plain way.
2. A simpler way to build the library `pm` or its module (`dataset`, `grammar`, `regexp`, `stream`) with all dependencies is to build a corresponding target with `cmake` in a plain way.
3. Modules `type_abuse` and `utils` are header-only and need not be built.

5.1 Plain build

Follow the instructions of your favourite `C++` compiler which meets all few requirements described in corresponding section. Do not forget to turn on features of 11 standard. Feed all `.cpp` and `.hpp` files to the compiler.

5.2 `cmake`-based build

Instruction tested for `cmake 3.5.2` and `gcc 4.5.8`. Based on documentation for `cmake`, the author believes that the earliest acceptable version of `cmake` is `3.1`.

Plain use of `cmake` generates a static library for `pm`, as well as a static standalone library for any its module with all dependencies. Using the instruction for `cmake`, build the target `pm` (the whole library), or one of the targets `dataset`, `grammar`, `regexp`, `stream` (corresponding module with depenencies).

Step-by-step instruction for `Linux` terminal

The following sequence of commands is to be written in `Linux` terminal to build a static version of the library `pm` in the folder `folder/build` having the archive `pm.1.0.tar.gz` in the folder `folder`. To build the library from another forlder or to another folder, replace the folder names. To build the library from unpacked sources (of subfolder `src`), skip the unpacking step, and replace the folder `../src` with root folder of source files. To build a module instead of the whole library, replace "`pm`" at the last step with the name of the module.

As a result, a static library file will be generated in the folder `folder/build`. The name of the file will follow conventions of your operating system. For instance, in `Linux` the name of the static library built with a target `<trg>` is `lib<trg>.a`.

The commands are:

1. Go to the archive folder.
```
cd folder
```
2. Unpack the archive to the current folder.
```
tar -xf pm.1.0.tar.gz
```
3. Create a subfolder `build`, if not created, and go to this subfolder.
```
mkdir build
cd build
```
4. Generate makefiles.
```
cmake ../src
```
5. Build the library.
```
make pm
```

6 Namespaces

The library is exactly a collection of all its modules, and nothing more. All definitions of the library are located in namespace `pm`.

All definitions of a module `M` are located in namespace `pm::M`. For instance, to access the class `Regexp` of the module `regexp`, one should write

`pm::regexp::Regexp`.

A module namespace may contain subspaces. For instance, some definitions of the module `regexp` related to parse context of a reversable stream are located in the namespace

`pm::regexp::context`.

Detailed description of interfaces provided by modules is placed in corresponding sections.

The manual follows several conventions:

1. By default, definitions of a module are located in the module namespace.
2. The namespace is omitted, if a component of an entity being defined is located in the same namespace as the entity.
3. Only the subspaces are written, if the component is located in some subspace of the namespace of the entity. For instance, if, say, class `T` is located in a namespace `a::b`, and some its subcomponent `X` — in a namespace `a::b::c::d`, then `c::d::X` is how the subcomponent is referred to in the definition of `X`.
4. Everything except "`pm::`" is written for entities from other modules.
5. The full namespace prefix is written for other entities.

7 Interface of `utils` module

All definitions are located in namespace

`pm::utils`.

7.1 `template<typename Element> class Numeration;`

Location

`utils/numeration.hpp`

Description

Class of numerations.
At each lifetime moment numeration defines a bijection between elements of type `Element` and a set $\{0, 1, \ldots, k-1\}$, where k is the number of elements to be mapped.
Preimage is any element to be mapped.
Image is a number to which a preimage is mapped.
Domain is the set of all preimages.
When an element is added to the domain, all images of existing preimages remain unchanged.
When an element is deleted from the domain, bijection may change in any way.

Template argument requirements

Objects of type `Element` should be comparable with `<`.
Depending on used methods, copy-constructor or move-constructor may be needed for type `Element`.

Default constructors and assignments, destructor

All are present, standard semantics.
`Numeration()` initializes numeration with an empty domain.

Object altering methods

`size_t add(const Element & el);`
Description. Add `el` to domain, if not present.
In any case return image of `el`.

`size_t add(Element && el);`
Description. Move-analogue of previous method.

`void add(const Numeration<Element> & num);`
Description. Add all preimages of `num` to domain of `*this`

`void add(Numeration<Element> && num);`
Description. Move-analogue of previous method.

`void clear();`
Description. Clean domain.

`void remove(const Element & el);`
Description. Delete `el` from domain.

Access methods

`const Element & element_at(size_t i) const;`
Description. Return preimage of `i`.
Requirements. Size of domain is greater or equal to `(i+1)`.

`bool has(const Element & el) const;`
Description. Return `true` $\Leftrightarrow$ domain contains `el`.

`size_t index_of(const Element & el) const;`
Description. Return image of `el`
Requirements. Domain contains `el`.

`size_t size() const;`
Description. Return size of domain.

`std::vector<Element>::const_iterator begin() const;`
Description. Return iterator to the first element of domain (for "`range-based for`").

`std::vector<Element>::const_iterator end() const;`
Description. Return iterator to the end of domain (for "`range-based for`").

7.2 `template<typename Value> class Optional;`

Location

`utils/optional.hpp`

Description

Class of options.
At any lifetime moment an option
- either nonempty, and contains (stores) a value of type `Value`,
- or empty, and does not contain any value.

Template argument requirements

Depending on used methods, copy-constructor or move-constructor may be required for type `Value`.

Default constructors and assignments, destructor

All are present, standard semantics.
`Optional()` initializes an empty option.

Other constructors and assignments

`Optional(const Value & value);`
Description. Initialize nonempty option containing a copy of `value`.

`Optional(Value && value);`
Description. Move-analogue of previous constructor.

`Optional & operator =(const Value & value);`
Description. Store copy of `value` into option.

`Optional & operator =(Value && value);`
Description. Move-analogue of previous assignment.

Object altering methods

`void unset();`
Description. Make option empty.

Access methods

`bool has_value() const;`
Description. Return `true` $\Leftrightarrow$ option is empty.

`Value & value();`
Description. Return stored value.
Requirements. Option is nonempty.

`const Value & value() const;`
Description. Const-analogue of previous method.

7.3 Functions fot `stl` containers

Location

`utils/container_functions.hpp`

Conventions

For all further template function prototypes,
- `Container` is a type for an `stl` contaner, or a similar container;
- `Element` is a type for elements of the container.

Functions

```
template<typename Container>
void append(Container & where, const Container & who);
```
Description. Append elements of `who` to the end of elements of `where`.
Requirements. Expression `where.append(where.end(), who.begin(), who.end())` is well-formed.

```
template<typename Container>
bool disjoint(const Container & X, const Container & Y);
```
Description. Return `true` $\Leftrightarrow$ containters `X` and `Y` do not contain `x`, `y` such that `x == y`.

Requirements. Type `Container`
- accepts "`range-based for`" for its elements, and
- contains the method `find(const Element &)`.

```
template<typename Container>
void erase_index(Container & container, size_t ind);
```
Description. Delete `ind`-th element from container, i.e. element referred with `container.begin() + ind`.
Requirements. Expression `container.begin() + ind` is well-formed.

```
template<typename Container>
bool intersects(const Container & X, const Container & Y);
```
Description. Return `true` ⇔ containers `X` and `Y` contain elements `x`, `y` such that `x == y`.
Requirements. Type `Container`
- accepts "`range-based for`" for its elements, and
- contains the method `find(const Element &)`.

```
template<typename Element, typename Container>
bool is_in(const Element & element, const Container & container);
```
Description. Return `true` ⇔ container contains element `x` such that `x == element`.
Requirements. Expression `container.find(element)` is well-formed.
Elements are comparable with `==`.

```
template<typename Element, typename Container>
bool isnt_in(const Element & element, const Container & container);
```
Description. Return `true` ⇔ containter does not contain elements `x` such that `x == element`.
Requirements. Expression `container.find(element)` is well-formed.
Elements are comparable with `==`.

```
template<typename Container>
void merge(Container & where, const Container & who);
```
Description. Insert elements of `who` into elements of `where`.
Requirements. Container containes methods `begin()`, `end()`, as well as a binary method `insert` with arguments — iterators pointing to begin and end of spectre of inserted elements.

8 Interface of `stream` module

All definitions are located in namespace `pm::stream`.

8.1 `class Stream;`

Location

`stream/stream.hpp`

Description

This is a class of reversible streams.
A reversible stream contains a standard input stream (related stream), and a string buffer of unlimited size. A reversible stream provides means for
- a read of symbols and strings analogous to the unformatted read of symbols and symbol sequences from the related stream, and
- a putback of symbols and strings analogous to one provided by the methods `unget` and `putback` of a standard input stream, but guaranteed to always successfully put the symbols back.

The buffer is split into blocks (stack of strings) of equal fixed size. The size may be chosen with a special constructor. The size does not affect functionality, it influences read/putback performance only.
Each read operation is split into two phases: the first phase is to read from the buffer untill there is nothing to read, or until the buffer is empty; if after the first phase these is still something to read, then symbols are being read from the related stream. The first stage is always successful. The whole read operation may be unsuccessful, and the only reason is that something went wrong in the related stream.
Symbols are being putback into the buffer only, and in such a way that a natural read order is preserved:
- returned symbols/strings are being read from last ones to first ones (fifo, stack);
- inside a returned string symbols are being read from first ones to last ones.

Putback operations are always successfull.

Default constructors and assignments, destructor

All such constructors and assignments are deleted.
The destructor is usual.

Other constructors

`Stream(std::istream & stream);`
Description. Make `stream` a related stream (via reference).
Initialize an empty buffer with blocks of size `100`.

`Stream(std::istream & stream, size_t buffer_size);`
Description. Make `stream` a related stream (via reference).
Initialize an empty buffer with blocks of size `buffer_size`.

Object altering methods

`bool get(char & c);`
Description. Try to read a symbol.
Upon success, write read symbol to `c`.
Otherwise, do not change `c`.
In any case return `true` $\Leftrightarrow$ a symbol is successfully read (and stored in `c`).

`size_t get(std::string & s, size_t size);`
Description. Read as much as possible, but not more than `size` symbols.
Write actually read symbol sequence into `s` (if no symbols are read, then clean `s`).
Return the number of read symbols.

`bool get_strict(std::string & s, size_t size);`
Description. Try to read exactly `size` symbols.
Upon success, write these symbols into `s`.
Otherwise, putback all actually read symbols and do not change `s`.
In any case return `true` $\Leftrightarrow$ all `size` symbols were successfully read (and stored in `s`).

`bool skip();`
Description. Try to read a symbol, and do not store this symbol.
Return `true` $\Leftrightarrow$ a symbol is successfully read.

`size_t skip(size_t size);`
Description. Try to read as much as possible, but not more than `size` symbols, and do not thore them.
Return the number of actually read symbols.

`bool skip_strict(size_t size);`
Description. Try to read exactly `size` symbols, and do not store them.
In case of fail, putback actually read symbols.
In any case return `true` $\Leftrightarrow$ all `size` symbols were successfully read.

`void unget(char c);`
Description. Putback a symbol `c`.

`void unget(const std::string & s);`
Description. Putback a string `s`.

Access methods

`bool finished() const;`
Description. Return `true` $\Leftrightarrow$ the buffer is empty, and the related stream is finished or broken, i.e. at least one of error flags is raised.[10] If any flag is raised, `*this` stops reading from the related stream.

9 Interface of `type_abuse` module

All definitions are located in namespace `pm::type_abuse`.

[10] For those who is out of context. There are exactly three error flags of a standard input stream: attempt to read beyond end of file; logical input/output error; read/write error during input/output. At least one raised flag usually means that it is a bad idea to continue reading from the stream.

9.1 General description

A little reminder. C++ has two popular concepts of pointers:

- plain pointer to type T, which is a pointer of type T *;
- smart pointer to type T, which is a general notion for standard pointer types with garbage collection and specific management features for multiple pointing to a single object, e.g. std::shared_ptr<T> and std::unique_ptr<T>.

This module provides alternative concept of pointers called "tricky pointers".[11]

9.1.1 Data blocks

A block (of data) is a special entity to which a tricky pointer points. Blocks may have different types of structure. Precise technical details of the structure are not discussed.

A block can be

- nonempty, and contain (store) data (value, object);
- empty, and contain nothing.

A data of a block is the data stored in the block.

A tricky pointer points to exactly one block at each moment of lifetime (owns this block; the block is an ownee of the pointer).

Conceptual difference between tricky pointers and both plain and smart pointers is that there exist no empty tricky pointers (such as nullptr), but there exist tricky pointers owning empty blocks. Unlimited number of distinct empty blocks may be created. An empty block may be owned by many tricky pointers. If some data is recorded to an empty block, then all owners gain access to that data.

An envelope of an object is a new nonempty block which stores the object.

An address block is a block which can provide plain pointer to stored object in case the block is nonempty.[12] That address is called a base of the block. The provided pointer has a certain type (say, T *), and the type T is called a base type of a block and related tricky pointers.

Base-envelope of an address is a new nonempty block which stores exactly the object pointed by the address.

Address blocks are well-suited for management of dynamically created objects of base and derived types, just like it happens in plain and smart pointers. Nonaddress blocks are not recommended for these purposes, their main design intention is to provide means to record data, delete data, and arbitrarily change type of data during block lifetime. Such kind of data — which has a certain type determined during creation, but arbitrary and potentially any, is called a "whatever data".

9.1.2 Garbage collection

Tricky pointers, just like smart ones, have embedded garbage collection:[13]

- if nobody owns a nonempty block, then data stored in the block is destructed (destroyed; freed);
- if operations say that data stored in a block should "disappear", then it is destructed.

In paricular, it means that stored data should not be explicitly or implicitly destructed by any means but provided by tricky pointers owning a block with corresponding data (e.g. to explicitly call a destructor, or free the memory allocated by data, or use one base in several blocks, or use a base in smart pointers, and so on).

The data of a nonempty address block is destructed by a call of a destructor of a dereference of a base of the block.

9.1.3 Block ownership

Ownership of a block by tricky pointers is shared between pointers of the same type via copy, move, assignment, and special methods of pointers according to documentation, and not in any other way.[14] An attempt to transfer ownership by any other (undocumented) means leads to an unspecified behavior — probably, to an unexpected data duplication with incorrect ownership transfer, or to segmentation fault due to access of data in deallocated memory.

9.1.4 Operations on blocks

To record value v to ablock means that block starts to store the value v (and the old value, if any, "disappears").

To insert a base (a value of a plain pointer) to an address block means that the value becomes a base of the block, and the block starts to store data pointed by this value.

To rewrite a block A with a block B means the following:

- if B is empty, then A becomes empty;
- if B is nonempty, then a copy of data stored if B is being recorded into A.

There exist three possibilities how data are copied:

1. Default case: allocation of memory, and copy-initialization or move-initialization;

[11] The name is inspired by "smart pointers". Such a dumb name is the result of lack of imagination.

[12] Just like the method get of smart pointers.

[13] To be more precise, tricky pointers use the garbage collection provided by std::shared_ptr and in some cases std::unique_ptr.

[14] Again, just like for smart pointers.

2. Explicit case: insertion of base which is obtained with a special method of a base class;
3. Explicit case: the existing object is rewritten via an assignment operator.

To clone a block is conceptually to do the following: to create a new empty block, and immediately rewrite this block by a cloned one. A clone of a block is the new block which is obtained by cloning.

9.1.5 Operations on tricky pointers

A redirection of a tricky pointer from one block to another means that the pointer loses ownership of the old block (this block remains unchanged, except garbage collected if needed) and establishes (shares) ownership of the new block.

To detach a tricky pointer is to redirect it to a clone of its current ownee.

A clone of a tricky pointer is a tricky pointer of the same type owning a clone of the ownee of the cloned pointer.

9.1.6 Tricky pointer diversity

Distinctions between types of tricky pointers may be summarized as follows:

1. are owned blocks address blocks;
2. are there any restrictions on a base type of address blocks;
3. is it possible to own an empty block;
4. can emptiness and stored data type be changed at any moment of a block lifetime, or once during block initialization only.

To choose a certain type of a tricky pointer, one may take into account following questions:

- what capabilities should be provided by a pointer (e.g. if it is absolutely necessary to process plain pointers to stored data, as well as to access overloaded virtual methods of derived classes via plain pointer to a base class, then tricky pointers to address blocks should be used);
- what performance overhead is acceptable (wide capabilites mean low performance).

9.2 `template<typename Base> class BRef;`

Location

`type_abuse/bref.hpp`

Description

This tricky pointer owns nonempty address blocks with base type `Base`.

Type of stored data is always `Base`, with the following exception. Values of derived types are allowed to be stored, if the following conditions hold:

1. Base insertion is the only way to record data to block.
2. Block base dereference destructor correctly destroys stored data.
3. No copies of stored data are made with methods of tricky pointer.

Template argument requirements

Depending on used methods, default constructor, copy-constructor, move-constructor, copy-assignment, move-assignemnt, and other constructors requested by user may be required for `Base`.

Default constructors and assignments, destructor

`BRef();`
Description. Direct `*this` to envelope of `Base()`.

`BRef(const BRef<Base> & ref);`
Description. Direct `*this` to ownee of `ref`.

`BRef(Bref<Base> && ref);`
Description. Move-analogue of previous constructor.

`BRef & operator =(const BRef<Base> & ref);`
Description. Redirect `*this` to ownee of `ref`.
Return `*this`.

`BRef & operator =(BRef<Base> && ref);`
Description. Move-analogue of previous assignment.

`~BRef();`
Description. Usual destructor.

Other constructors

`BRef(Base * ptr);`
Description. If `ptr == nullptr`, then direct `*this` to envelope of `Base()`.
Otherwise direct `*this` to base-envelope of `ptr`.
Requirements. `ptr` is dereferenceable.
Data pointed by `ptr` should not be deleted, directly or indirectly, by anyone except owners of the new envelope.

`BRef(const Base & v);`
Description. Direct `*this` to envelope of copy of `v`.

`BRef(Base && v);`
Description. Move-analogue of previous constructor.

Other operators

`Base * operator ->() const;`
Description. Analogue of the same operator for plain pointers.
Access operator for fields and methods of dereference of data of ownee of `*this`.

`Base & operator *() const;`
Description. Analogue of the same operator for plain pointers.
Return dereference of data of ownee of `*this`.

Object altering methods

`void detach();`
Description. Detach `*this`.

`void detach_own(Base * ptr);`
Description. Redirect `*this` to base-envelope of `ptr`.
Requirements. `ptr` is dereferenceable.
Data pointed by `ptr` should not be deleted, directly or indirectly, by anyone except owners of the new envelope.

`void detach_receive(const BRef<Base> & ref);`
Description. Redirect `*this` to clone of ownee of `ref`.

```
template<typename ...  Args>
detach_set(Args && ...  args);
```
Description. Redirect `*this` to envelope of `Base(args ...)`.

Object preserving, contents altering methods

`void receive(const BRef<Base> & ref) const;`
Description. Rewrite ownee of `*this` with ownee of `ref` via assignment.

```
template<typename ...  Args>
void set(Args && ...  args) const;
```
Description. Record `Base(args ...)` to ownee of `*this` via assignment.

Object and contents preserving methods

`BRef<Base> clone() const;`
Description. Return clone of `*this`.

`void send(const BRef<Base> & ref) const;`
Description. Rewrite ownee of `ref` with ownee `*this` via assignment.

Access methods

`Base * get() const;`
Description. Return base of ownee of `*this`.
Requirements. Data pointed by returned value should not be deleted, directly or indirectly, by anyone except owners of ownee of `*this`.

`Base * get_copy() const;`
Description. Create copy of data of ownee of `*this`, and return pointer to that copy.

`Base & val() const;`
Description. Return dereference of base of ownee of `*this`.

Static methods

```
template<typename ... Args>
static BRef<Base> create(Args && ... args);
```
Description. Return tricky pointer owning envelope of `Base(args ...)`.

9.3 `class CDRef;`

Location

`type_abuse/cdref.hpp`

Description

This tricky pointer owns empty and nonempty nonaddress blocks.
Emptiness of the block and type of stored data are determined once during block initialization and remain unchanged since.

Requirements for types of stored data

Depending on used methods, copy-constructor, move-constructor, and other constructors may be required for stored data type.

Default constructors and assignments, destructor

`CDRef();`
Description. Direct `*this` to new empty block.

`CDRef(const CDRef & ref);`
Description. Direct `*this` to ownee of `ref`.

`CDRef(CDRef && ref);`
Description. Move-analogue of previous constructor.

`CDRef & operator =(const CDRef & ref);`
Description. Redirect `*this` to ownee of `ref`.
Return `*this`.

`CDRef & operator =(CDRef && ref);`
Description. Move-analogue of previous assignment.

`~CDRef();`
Description. Usual destructor.

Other operators

`operator bool() const;`
Description. Return `true` $\Leftrightarrow$ ownee of `*this` is empty.

Object altering methods

`void detach();`
Description. Detach `*this`.

`void detach_receive(const CDRef & ref);`
Description. Redirect `*this` to clone of ownee of `ref`.

```
template<typename Value, typename ... Args>
void detach_set(Args && ... args);
```
Description. Redirect `*this` to envelope of `Value(args ...)`.

Object preserving methods

`CDRef clone() const;`
Description. Return clone of `*this`.

Access methods

`bool empty() const;`
Description. Return `true` ⇔ ownee of `*this` is empty.

`bool nonempty() const;`
Description. Return `true` ⇔ ownee of `*this` is nonempty.

```
template<typename Value>
Value & val() const;
```
Description. Return data of ownee of `*this` assumed to be of type `Value`.
Requirements. Ownee of `*this` is nonempty and stored data of type `Value`.

Static methods

```
template<typename Value, typename ... Args>
static CDRef create(Args && ... args);
```
Description. Return tricky pointer owning envelope of `Value(args ...)`.

9.4 `template<typename Base> class CRef;`

Location

`type_abuse/cref.hpp`

Description

This tricky pointer owns empty and nonempty address blocks with base type `Base`.
Emptiness of the block and type of stored data are determined once during block initialization and remain unchanged since.

Template argument requirements

Depending on used methods, copy-constructor, move-constructor, and other constructors requested by user may be required for `Base`.
Data copy is based on a cloning method of `Base`: the method `Base * Base::clone()` which returns plain pointer to new copy of object for which the method was called.
If copy of data of derived types if required, then usually it means that the cloning method should be virtual and redefined in derived classes.

Default constructors and assignments, destructor

`CRef();`
Description. Direct `*this` to new empty block.

`CRef(const CRef<Base> & ref);`
Description. Direct `*this` to ownee of `ref`.

`CRef(CRef<Base> && ref);`
Description. Move-analogue of previous constructor.

`CRef & operator =(const CRef<Base> & ref);`
Description. Redirect `*this` to ownee of `ref`.
Return `*this`.

`CRef & operator =(CRef<Base> && ref);`
Description. Move-analogue of previous assignment.

`~CRef();`
Description. Usual destructor.

Other constructors

`CRef(Base * ptr);`
Description. Direct `*this` to base-envelope of `ptr`.
Requirements. `ptr` is dereferenceable.
Data pointed by `ptr` should not be deleted, directly or indirectly, by anyone except owners of the new envelope.

Other operators

`Base * operator ->() const;`
Description. Analogue of the same operator for plain pointers.
Access operator for fields and methods of dereference of data of ownee of `*this`.
Requirements. Ownee of `*this` is nonempty.

`Base & operator *() const;`
Description. Analogue of the same operator for plain pointers.
Return dereference of data of ownee of `*this`.
Requirements. Ownee of `*this` is nonempty.

`operator bool() const;`
Description. Return `true` ⇔ ownee of `*this` is nonempty.

Object altering methods

`void detach();`
Description. Detach `*this`.

`void detach_own(Base * ptr);`
Description. Redirect `*this` to base-envelope of `ptr`.
Requirements. `ptr` is dereferenceable.
Data pointed by `ptr` should not be deleted, directly or indirectly, by anyone except owners of the new envelope.

`void detach_receive(const CRef<Base> & ref);`
Description. Redirect `*this` to clone of ownee of `ref`.

```
template<typename Value = Base, typename ...  Args>
void detach_set(Args && ...  args);
```
Description. Redirect `*this` to envelope of `Value(args ...)`.

Object preserving methods

`CRef<Base> clone() const;`
Description. Return clone of `*this`.

Access methods

`bool empty() const;`
Description. Return `true` ⇔ ownee of `*this` is empty.

`Base * get() const;`
Description. Return base of ownee of `*this`.
Requirements. Ownee of `*this` is nonempty.
Data pointed by returned address should not be deleted, directly or indirectly, by anyone except owners of ownee of `*this`.

`Base * get_copy() const;`
Description. Create copy of data of ownee of `*this`, and return plain pointer to that copy.
Requirements. Ownee of `*this` is nonempty.

`bool nonempty() const;`
Description. Return `true` ⇔ ownee of `*this` is nonempty.

```
template<typename Value = Base>
Value & val() const;
```
Description. Return data of ownee of `*this`, casted to type `Value` via `static_cast`.
Requirements. Ownee of `*this` is nonempty.
Cast is correct.

Static methods

```
template<typename Value = Base, typename ...  Args>
static CRef<Base> create(Args && ...  args);
```
Description. Return tricky pointer to envelope of `Value(args ...)`.

9.5 `class DRef;`

Location

`type_abuse/dref.hpp`

Description

Class of tricky pointers to whatever data.

This pointer owns empty and nonempty nonaddress blocks. Emptiness and stored data type can be freely changed during lifetime.

Requirements for types of stored data

Depending on used methods, copy-constructor, move-constructor, and other constructors requested by user may b required.

Default constructors and assignments, destructor

`DRef();`
Description. Direct `*this` to new empty block.

`DRef(const DRef & ref);`
Description. Direct `*this` to ownee of `ref`.

`DRef(DRef && ref);`
Description. Move-analogue of previous constructor.

`DRef & operator =(const DRef & ref);`
Description. Redirect `*this` to ownee of `ref`.
Return `*this`.

`DRef & operator =(DRef && ref);`
Description. Move-analogue of previous assignment.

`~DRef();`
Description. Usual destructor.

Other operators

`operator bool() const;`
Description. Return `true` $\Leftrightarrow$ ownee of `*this` is nonempty.

Object altering methods

`void detach();`
Description. Detach `*this`.

`void detach_receive(const DRef & ref);`
Description. Redirect `*this` to clone of ownee of `ref`.

```
template<typename Value, typename ...  Args>
void detach_set(Args && ...  args);
```
Description. Redirect `*this` to envelope of `Value(args ...)`.

`void detach_unset();`
Description. Redirect `*this` to new empty block.

Object preserving, contents altering methods

`void receive(const DRef & ref) const;`
Description. Rewrite ownee of `*this` with ownee of `ref`.

```
template<typename Value, typename ...  Args>
void set(Args && ...  args) const;
```
Description. Record `Value(args ...)` to ownee of `*this`.

`void swap(const DRef & ref) const;`
Description. Switch data stored in ownees of `*this` and `ref`.

`void unset() const;`
Description. Make ownee of `*this` empty.

Object and contents preserving methods

`DRef clone() const;`
Description. Return clone of `*this`.

`void send(const DRef & ref) const;`
Description. Rewrite ownee of `ref` with ownee of `*this`.

Access methods

`bool empty() const;`
Description. Return `true` ⇔ ownee of `*this` is empty.

`bool nonempty() const;`
Description. Return `true` ⇔ ownee of `*this` is nonempty.

```
template<typename Value>
Value & val() const;
```
Description. Return data of ownee of `*this` assumed to be of type `Value`.
Requirements. Ownee of `*this` is nonempty and stores data of type `Value`.

Static methods

```
template<typename Value, typename ... Args>
static DRef create(Args && ... args);
```
Description. Return tricky pointer owning envelope of `Value(args ...)`.

9.6 `template<typename Base> class Ref;`

Location

`type_abuse/ref.hpp`

Description

This tricky pointer owns empty and nonempty address blocks with base type `Base`.
Emptiness and stored data type may be freely changed during block lifetime.

Template argument requirements

Depending on used methods, copy-constructor, move-constructor, and other constructors requested by user may be required for stored data type.

Copy of data is based on a cloning method of `Base`: the method `Base * Base::clone()` which returns a plain pointer to a new copy of object for which the method was called.

If copy of objects of derived types is required, then it usually means that the cloning method is virtual and redefined in derived classes.

Default constructors and assignments, destructor

`Ref();`
Description. Direct `*this` to new empty block.

`Ref(const Ref<Base> & ref);`
Description. Direct `*this` to ownee of `ref`.

`Ref(Ref<Base> && ref);`
Description. Move-analogue of previous constructor.

`Ref & operator =(const Ref<Base> & ref);`
Description. Redirect `*this` to ownee of `ref`.
Return `*this`.

`Ref & operator =(Ref<Base> && ref);`
Description. Move-analogue of previous assignment.

`~Ref();`
Description. Usual destructor.

Other constructors

`Ref(Base * ptr);`
Description. Direct `*this` to base-envelope of `ptr`.
Requirements. `ptr` is dereferenceable.
Data pointed by `ptr` should not be deleted, directly or indirectly, by anyone except owners of the new envelope.

Other operators

`Base * operator ->() const;`
Description. Analogue of the same operator for plain pointers.
Access operator for fields and methods of dereference of data of ownee of `*this`.
Requirements. Ownee of `*this` is nonempty.

`Base & operator *() const;`
Description. Analogue of the same operator for plain pointers.
Return dereference of data of ownee of `*this`.
Requirements. Ownee of `*this` is nonempty.

`operator bool() const;`
Description. Return `true` $\Leftrightarrow$ ownee of `*this` is nonempty.

Object altering methods

`void detach();`
Description. Detach `*this`.

`void detach_own(Base * ptr);`
Description. Redirect `*this` to base-envelope of `ptr`.
Requirements. `ptr` is dereferenceable.
Data pointed by `ptr` should not be deleted, directly or indirectly, by anyone except owners of the new envelope.

`void detach_receive(const Ref<Base> & ref);`
Description. Redirect `*this` to clone of ownee of `ref`.

```
template<typename Value = Base, typename ... Args>
void detach_set(Args && ... args);
```
Description. Redirect `*this` to envelope of `Value(args ...)`.

`void detach_unset();`
Description. Redirect `*this` to new empty block.

Object preserving, contents altering methods

`void own(Base * ptr) const;`
Description. Insert base `ptr` into ownee `*this`.

`void receive(const Ref<Base> & ref) const;`
Description. Rewrite ownee of `*this` with ownee of `ref`.

```
template<typename Value = Base, typename ... Args>
void set(Args && ... args) const;
```
Description. Record `Value(args ...)` to ownee of `*this`.

`void swap(const Ref<Base> & ref) const;`
Description. Switch data stored in ownees of `*this` and `ref`.

`void swap(std::unique_ptr<Base> & ptr) const;`
Description. Switch data stored in ownees of `*this` and `ptr`.

`void unset() const;`
Description. Make ownee of `*this` empty.

Object and contents preserving methods

`Ref<Base> clone() const;`
Description. Return clone of `*this`.

Access methods

`bool empty() const;`
Description. Return `true` ⇔ ownee of `*this` is empty.

`Base * get() const;`
Description. Return base of ownee of `*this`.
Requirements. Ownee of `*this` is nonempty.
Data pointed by returned address should not be deleted, directly or indirectly, by anyone except owners of ownee of `*this`.

`Base * get_copy() const;`
Description. Create copy of data of ownee of `*this`, and return plain pointer to that copy.

`bool nonempty() const;`
Description. Return `true` ⇔ ownee of `*this` is nonempty.

```
template<typename Value = Base>
Value & val() const;
```
Description. Return data of ownee of `*this` assumed to be of type `Value`.
Requirements. Ownee of `*this` is nonempty and stores data of type `Value`.

Static methods

```
template<typename Value = Base, typename ...  Args>
static Ref<Base> create(Args && ...  args);
```
Description. Return tricky pointer owning envelope of `Value(args ...)`.

9.7 Function type for whatever data modification

Location

`type_abuse/dfun.hpp`

Common remark

Further description includes intentions of designed definitions, which flooded the author's head when everything was coded. These intentions may be followed, or not followed, as you wish.

Types

`typedef std::function<void(const DRef &, const DRef &)> Applier;`
Intention. Data stored in the second argument are applied to data stored in first argument — e.g., is added, provides change conditions, recorded into field, an so on.

`typedef std::function<DRef()> Creator;`
Intention. Plain creation of new data blocks, i.e. generation of data out of nowhere.

`typedef std::function<void(const DRef &)> Modifier;`
Intention. User-defined modification of data stored in the argument.

10 Interface of `regexp` module

All definitions are located in namespace

`pm::regexp`.

10.1 General description

10.1.1 What is regular expression

The notion of regular expressions is well-known, at least to those who had the need to parse a string, or a stream. The author tried to write this manual as understandable as possible, even for those who had no such needs (but know what a mathematics is). The initial explanation point is a regular expression in mathematics.

A regular expression r — is a syntactic object defined by the following Backus-Naur form (elements of the right side are quoted to reduce ambiguity):

$$r \texttt{ ::= } ``\emptyset'' \texttt{ | } ``\lambda'' \texttt{ | } ``a'' \texttt{ | } ``(r|r)'' \texttt{ | } ``(r \cdot r)'' \texttt{ | } (r^*)$$

Here a is a symbol of a predefined finite alphabet A, i.e. of a finite set of symbols, and λ is an empty string. Brackets may be omitted according to precedence: *, then $\cdot$, then $|$.

Components of an expression can be split into two parts: basic expressions (an empty set $\emptyset$, an empty string λ, a symbol a), and compositional operations (an alternative $|$, a concatenation $\cdot$, a Kleene star *). Syntax (structure) of a regular expression defines a structure of strings which the expression recognizes (accepts).

A meaning (semantics) of an expression r is a language $L(r)$ accepted by the expression, i.e. the a set of strings accepted by it. $\emptyset$ accepts no strings. λ accepts an empty string (only). a accepts a string consisting of one symbol a. $r_1|r_2$ accepts strings accepted by at least one of subexpressions r_1, r_2. $r_1 \cdot r_2$ accepts strings w_1w_2, where w_1 is accepted by r_1, and w_2 is accepted by r_2. r^* accepts strings $w_1w_2 \dots w_k$, where each w_i is accepted by r.

For example, the expression $(0|1)\cdot(0|1)^*$ accepts exactly all nonempty strings of 0-s and 1-s, i.e. all binary encodings of numbers with possible insignificant 0-s.

Regular expressions in programming usually differ from mathematical ones: have a wider semantics (more basic expressions, and more operations), and narrower semantics (an expression accepts less strings than it accepts in mathematical sense).

The most popular use of regular expressions in programming is as follows: given a string h, to find out if it is accepted, and if yes, then perform some actions according to the structure of the string, i.e. to parse the string. If a parse is successfull, the string is accepted (and actions are performed), otherwise rejected. A flow of the parse is defined by parsing algoritms of basic expressions, and by algorithms of operations which dictate an order of subexpression execution and an acceptance criterion.

For example, the algorithm of

- $r_1|r_2$ is to execute r_1, and in case of failure to execute r_2; in case of at least on success a word accepted by a successfull subexpression execution is accepted (and the word is rejected otherwise);
- $r_1 \cdot r_2$ is to execute r_1, and in case of success to execute r_2 for a substring starting with the symbol next to last accepted by r_1; in case both subexpression executions were successfull, a concatenation of accepted words;
- r^* executes over and over until the first fail; a concatenation of words accepted during executions is accepted.

The next question is what to do if we want not only to accept a word, but to obtain some useful information about the word if accepted. For instance, it would be useful not only to recognize that the word is binary encoding of a number, but to obtain the number itself. Every developer of regular expressions has his own answer. The author suggests to widen syntax and write something like that:

$$f(0|1) \cdot g(0|1)^*.$$

$f()$ means "save a digit whose symbol was accepted by the argument", and $g()$ — "multiply saved number by 2 and add a digit whose symbol was accepted by the argument". After the parse we get what we wanted in the saved value. $f()$, $g()$ and similar constructs are what we call functional operations over regular expression.

10.1.2 Regular expressions in `regexp`

Syntax of regular expressions in `regexp`includes a wide spectre of basic expressions, compositional operations, and functional operations which operate according to programming semantics rather than mathematical one. To work with regular expressions, use the non-template class `Regexp` and interfaces for basic expressions and operations.

Class `Regexp` allows to parse three types of objects with `match` methods:[15] strings, standard input streams, and reversable streams (`stream::Stream`).

Along with the result of the parse, which is either that a string is accepted, or rejected, a user (if needed) may obtain an accepted string itself (in an ownee of `type_abuse::BRef<std::string>` tricky pointer), and whatever data was generated (`type_abuse::DRef`). The need is expressed by a special argument of `match` methods, which is a parse context. Optionally, a parse context may be omitted, which would mean that no string and no data are needed.[16]

Basic expressions generate initial whatever data, which is either nothing, or an accepted symbol, or an accepted string. Compositional operations execute their subexpressions (i.e. call their `match` methods), including data processing, in a specific way. A functional operation contains exactly one subexpression, and several user-defined functions over whatever data (`type_abuse::Applier`, `type_abuse::Creator`, `type_abuse::Modifier`). All it does is executes the subexpression, and apply functions in a certain way before and after the execution.

10.1.3 Parse context

Each `match` method receives two arguments. The first argument is a parsed object. The second argument (which may be omitted) is a parse context (class `Context`). A parse context contains 4 tricky pointers:

1. A tricky pointer to a string accepted by the moment.
   Type `type_abuse::BRef<std::string>`.
2. A tricky pointer to whatever data generated by the moment.
   Type `type_abuse::DRef`.
3. A tricky pointer to an activity flag for the string.
   Type `type_abuse::BRef<bool>`.

[15] To be more precise, these are methods of `Literal`

[16] To be more precise, in this case a default context is sent which says that no string and no data are needed. This may be changed by user, but not recommended due to errors which may occur because of forgotten cleaning of these string and data between several calls of `match` methods.

4. A tricky pointer to an activity flag for the data.
   Type `type_abuse::BRef<bool>`.

If a string-activity flag is raised, then an accepted string is appended to the end of a string stored in a string pointer at the beginning of the parse. In all other cases (the parse is failed, or the flag is lowered), the stored string remains unchanged.

If a data-activity data is raised, and a parse is successfull, then generated data are changed according to the executed regular expression. In all other cases (the parse is failed, or the flag is lowered), the data remains unchanged.

If a context is not sent as the second argument, then the default context `default_context` is assumed. At the beginning of a program execution the default context contains an empty string, empty data, and lowered flags. These contents may be changed in the program, though it is not recommended.

When copied or moved, as well as when special methods are called, a context shares ownership of its tricky pointers. Any duplication of pointer data is made in an explicit way with special methods of a context, and of tricky pointers.

10.1.4 How to add lots of regular expressions at once

All ready-to-use basic regular expressions can be added at once with the header `regexp/expr_basic.hpp`.
All ready-to-use compositional operations can be added at once with the header `regexp/expr_composition.hpp`.
All ready-to-use functional operations can be added at once with the header `regexp/expr_function.hpp`.
All ready-to-use expressions and operations can be added at once with the header `regexp/expressions.hpp`

10.1.5 How to write new regular expressions and operations

A user can easily implement his own regular expression. To do it, one should:
1. Implement a class derived from `Literal`.
2. Overload the special method `match_full`, which is a full-match contexted method for reversable streams.
3. Implement any means to obtain an object of type `Regexp`, which stores a tricky pointer to the implemented literal object, in a way similar to what is provided by ready-to-use expressions and operations.

For details one may look into the files `regexp/basic/let.hpp`, and `regexp/basic/let.cpp`, which implement a basic regular expression "a given symbol".

10.1.6 General conventions on parsing of reversable stream

In general, a user can implement a regular expression which changes a stream and a context in any preferred way. However, such a freedom leads to errors and unexpected results. For this reason we propose several conventions on how "good" regular expressions should operate.

Besides the global classification (basic expressions, compositional operations, functional operations), each expression has a sort: either precise, or imprecise one. A precise regular expression tries to read not more than a certain number of symbols from a stream. The number of symbols read by an imprecise expression is hard to specify.

Any regular expression obtained with a compositional operation is imprecise by definition: subexpressions are executed, symbols are returned, and again, and again — so it is a rather complicated symbol-reading flow.

Any regular expression obtained with a functional operation is as precise as its subexpression: functional operations does not change the reading flow of their subexpressions, the only change is always successful processing of generated data.

Basic expressions can have any sort.

Any regular expression should follow several restrictions:
1. The full match method returns `true` $\Leftrightarrow$ the parse was successful.
2. All tricky pointers of the context point do not change their ownees after the parse as comparing to before the parse.
3. If the parse is successfull, then exactly the accepted string is being read from the reversable stream. If more symbols were read than needed, then the rest is putback to the stream as if were never read.
4. If the string flag was raised at the start of the successfull parse, then the contents of the string pointer is the initial string with appended accepted string. In all other cases (failed parse, or lowered flag) the contents of the string pointer before and after the parse are equal.
5. If the data flag was raised at the start of the successfull parse, then the contents of the data pointer is changed according to the regular expression. In all other cases (failed parse, or lowered flag) the contents of the data pointer before and after the parse are equal.
6. In any case the values of any activity flag before and after the parse are equal.

Interface description for regular expressions implicitly assumes all the restrictions, and thus the only information to be provided is:
1. A sort of a basic expression.
2. A number of symbols to be read by a precise basic expression.
3. A description of a string to be accepted by a basic expression.
4. Specifics of subexpression calls and an acceptance criterion for compositional operations.
5. Specifics of data generation.

10.2 `class Context;`

Location

`regexp/context.hpp`

Description

Class of parse contexts.
A parse context contains two fields (subcontexts):

1. An activational context `asc` for accepted string.
2. An activational context `adc` for generated data.

Together with internal structure of subcontexts, the context contains 4 tricky pointers:

1. A pointer to an accepted string.
2. A pointer to an activity flag for the string.
3. A pointer to a generated data.
4. A pointer to an activity flag for the data.

Default constructors and assignments, destructor

All are present, standard semantics up to default actions for tricky pointers.
`Context()` initializes pointers with new blocks which store an empty string, an empty data, and lowered flags.

Other constructors

`Context(const context::AString & asc, const context::AData & adc);`
Description. Direct pointers of `*this` to ownees of pointers of corresponding arguments.

`Context(const context::AString & asc, context::AData && adc);`

`Context(context::AString & asc, const context::AData & adc);`

`Context(context::AString & asc, context::AData && adc);`
Description. Move-analogues of previous constructor.

Object altering methods

`void detach();`
Description. Redirect pointers of `*this` to clones of their ownees.

`void detach_receive(const Context & sdc);`
Description. Redirect pointers of `*this` to clones of ownees of corresponding pointers of `sdc`.

Object preserving, contents altering methods

`void receive(const Context & sdc) const;`
Description. Rewrite ownees of pointers of `*this` with ownees of corresponding pointers of `sdc`.

Object and contents preserving methods

`Context clone() const;`
Description. Return context whose pointers own clones of ownees of corresponding pointers of `*this`.

`void send(const Context & sdc) const;`
Description. Rewrite ownees of pointers of `sdc` with ownees of corresponding pointers of `*this`.

Access methods

```
template<typename Value>
Value & data() const;
```
Description. Return generated data assumed to be of type `Value`.
Requirements. Ownee of pointer to generated data is nonempty and stores data of type `Value`.

`bool is_active() const;`
Description. Return `true` $\Leftrightarrow$ at least one of flags is raised.

`bool is_empty() const;`
Description. Return `true` $\Leftrightarrow$ ownee of pointer to generated data is empty.

`bool is_inactive() const;`
Description. Return `true` $\Leftrightarrow$ both flags are lowered.

`bool is_nonempty() const;`

Description. Return `true` ⇔ ownee of pointer to generated data is nonempty.

`std::string & string() const;`
Description. Return accepted string.

Fields

`context::AString asc;`
Description. Subcontext which contains pointer to accepted string, and pointer to activity flag for generated string.

`context::AData adc;`
Description. Subcontext which contains pointer to generated data, and pointer to activity flag for generated data.

10.3 `class Literal;`

Location

`regexp/literal.hpp`

Description

Class of literals.
It is one of main classes for regular-expression machinery, along with `Regexp`.
Literal is an object of `Literal` of any derived class.
This class contains parse methods for strings, standard input streams, and reversable streams — contexed and uncontexed (which means the default context).
These methods eventually call a full parse method of the class (`match_full`; for reversable streams; contexted) and return its result.
A full parse method of `Literal` immediately successfully returns.

Default constructors and assignments, destructor

All are present, standard semantics.
Destructor is virtual.

Methods

`bool match(stream::Stream & s, Context & context = default_context) const;`
Description. Call `match_full` with same arguments, return its result.

`bool match(std::istream & s, Context & context = default_context) const;`
Description. Call `match` with `stream::Stream(s)` and the same context, return its result.

`bool match(const std::string & s, Context & context = default_context) const;`
Description. Call `match` with base of `std::stringstream(s)` and the same context, return its result.

`virtual bool match_full(stream::Stream & s, Context & context = default_context) const;`
Description. Immediately return `true`.
The main intention of the method is to be overloaded in derived classes.

10.4 `class Regexp : public Literal;`

Location

`regexp/regexp.hpp`

Description

Class of regular expressions.
It is one of main classes for regular-expression machinery, along with `Literal`.
Inherits all parse methods of `Literal`, except the overloaded virtual method of full parse.
Full parse method redirects parsing to a contained literal, which is a data of an ownee of the only field — a tricky pointer (the pointer).

Default constructors and assignments, destructor

All are present, standard semantics up to default actions for tricky pointers.
`Regexp()` initializes a regular expression containing a trivial literal `lskip`.

Other constructors

`Regexp(const type_abuse::CRef<Literal> & rl);`
Description. Direct the pointer to ownee of `rl`.

`Regexp(type_abuse::CRef<Literal> && rl);`
Description. Move-analogue of previous constructor.

Object preserving methods

`match_full(stream::Stream & stream, Context & context) const override;`
Description. Call full parse method for contained literal.
Return result of the call.

Fields

`type_abuse::CRef<Literal> rl;`
Description. Tricky pointer to literal which contains full parse method called by `*this`.

10.5 `class context::Activator : public type_abuse::BRef<bool>;`

Location

`regexp/context/activator.hpp`

Namespace of the definiton

`pm::regexp::context`

Description

Class of activators.
An activator is a tricky pointer to a boolean value, assumed to be an activity flag (of something not contained in the activator).
The flag is raised, if it equald `true`, and lowered, if it equals `false`.
To raise the flag is to record `true` to the ownee of the activator.
To lower the flag is to record `false` to the ownee of the activator.

Interface inheritance

Inherits all interface of `type_abuse::BRef<bool>`.

Default constructors and assignments, destructor

`Activator()` is deleted.
Other constructors, assignments, and a destructor are present, and have the same semantics as in `type_abuse::BRef<bool>`.

Other constructors

`Activator(bool act);`
Description. Direct `*this` to envelope of `act`.

Object altering methods

`void detach_activate();`
Description. Redirect `*this` to envelope of `true`.

`void detach_deactivate();`
Description. Redirect `*this` to envelope of `false`.

Object preserving, contents altering methods

`void activate() const;`
Description. Raise flag.

`void deactivate() const;`
Description. Lower flag.

Object and contents preserving methods

`Activator clone() const;`
Description. Return clone of `*this`.

10.6 `class context::AData;`

Location

`regexp/context/adata.hpp`

Namespace of the definiton

`pm::regexp::context`

Description

Class of subcontexts used, among others, as fields of a parse contexts.
This subcontext contains an activator (a tricky pointer to a boolean flag), and a tricky pointer to whatever data. The data is active, if the flag is raised, and inactive otherwise.

Default constructors and assignments, destructor

`AData()` is deleted.
Other constructors, assignments, and destructor are present, standart semantics up to default actions for tricky pointers.

Other constructors

`AData(bool act);`
Description. Direct activator to envelope of `act`, and data pointer — to new empty block.

`AData(const Activator & ac, const type_abuse::DRef & dc);`
Description. Direct pointers to ownees of corresponding arguments.

`AData(const Activator & ac, type_abuse::DRef && dc);`

`AData(Activator && ac, const type_abuse::DRef & dc);`

`AData(Activator && ac, type_abuse::DRef && dc);`
Description. Move-analogues of previous constructor.

Object altering methods

`void detach();`
Description. Detach all pointers.

`void detach_receive(const AData & adc);`
Description. Redirect pointers to clones of ownees of corresponding pointers of `adc`.

```
template<typename Value, typename ...  Args>
void detach_set_activate(Args && ...  args));
```
Description. Redirect activator to envelope of `true`, and data pointer — to envelope of `Value(args ...)`.

```
template<typename Value, typename ...  Args>
void detach_set_deactivate(Args && ...  args));
```
Description. Redirect activator to envelope of `false`, and data pointer — to envelope of `Value(args ...)`.

`void detach_unset_activate();`
Description. Redirect activator to envelope of `true`, and data pointer — to new empty block.

`void detach_unset_deactivate();`
Description. Redirect activator to envelope of `false`, and data pointer — to new empty block.

Object preserving, contents altering methods

`void activate() const;`
Description. Raise flag.

`void deactivate() const;`
Description. Lower flag.

`void receive(const AData & adc) const;`
Description. Rewrite ownees of pointers with ownees of corresponding pointers of `adc`.

`template<typename Value, typename ... Args>`
`void set_activate(Args && ... args) const;`
Description. Raise flag and record `Value(args ...)` to ownee of value pointer.

`template<typename Value, typename ... Args>`
`void set_deactivate(Args && ... args) const;`
Description. Lower flag and record `Value(args ...)` to ownee of value pointer.

`void unset_activate() const;`
Description. Raise flag and make ownee of data pointer empty.

`void unset_deactivate() const;`
Description. Lower flag and make ownee of data pointer empty.

Object and contents preserving methods

`AData clone() const;`
Description. Return subcontext whose pointers own clones of ownees of corresponding pointers of `*this`.

`void send(const AData & sdc) const;`
Description. Rewrite ownees of pointers of `sdc` with ownees of corresponding pointers of `*this`.

Access methods

`template<typename Value>`
`Value & data() const;`
Description. Return stored data assumed to be of type `Value`.
Requirements. Ownee of data pointer is nonempty, and stores value of type `Value`.

`bool is_active() const;`
Description. Return `true` ⇔ data is active.

`bool is_empty() const;`
Description. Return `true` ⇔ ownee of data pointer is empty.

`bool is_inactive() const;`
Description. Return `true` ⇔ data is inactive.

`bool is_nonempty() const;`
Description. Return `true` ⇔ ownee of data pointer is nonempty.

Fields

`Activator ac;`
Description. Activator (tricky pointer to boolean flag) of stored data.

`type_abuse::DRef dc;`
Description. Tricky pointer to stored whatever data.

10.7 `class context::AString;`

Location

`regexp/context/astring.hpp`

Namespace of the definiton

`pm::regexp::context`

Description

Class of contexts used, among others, as fields of parse contexts.
This context contains an activator (a tricky pointer to a boolean flag), and a tricky pointer to a string (when used in a parse context, to an accepted string).
The string is active, if the flag is raised, and inactive otherwise.

Definition

`typedef AVal<std::string> AString`

10.8 template<typename Value> class context::AVal;

Location

regexp/context/aval.hpp

Namespace of the definiton

pm::regexp::context

Description

An object of type AVal<T> contains an activator, and a tricky pointer to a value of type T (a value pointer).
A value is a data stored in the ownee of the value pointer. A flag is a value stored in the ownee of the activator as a tricky pointer.
AVal<std::string>, among others, is used as a field of a parse context (to an accepted string and its activity flag).
The value is active, if the flag is raised, and inactive otherwise.

Template argument requirements

Depending on used methods, default constructor, copy-constructor, move-constructor, copy-assignemnt operator, move-assignment operator, and other constructors requested by a user may be required.

Default constructors and assignments, destructor

AVal() is deleted.
Other constructors, assignments, and destructor are present, standart semantics up to default actions for tricky pointers.

Other constructors

AVal(bool act);
Description. Direct activator to envelope of act, and value pointer — to envelope of Value().

AVal(bool act, const Value & v);
Description. Direct activator to envelope of act, and value pointer — to envelope of copy of v.

AVal(bool act, Value && v);
Description. Move-analogue of previous constructor.

AVal(const Activator & ac, const type_abuse::BRef<Value> & vc);
Description. Direct activator and value pointer to ownees of corresponding arguments.

AVal(const Activator & ac, type_abuse::BRef<Value> && dc);

AVal(Activator && ac, const type_abuse::BRef<Value> & dc);

AVal(Activator && ac, type_abuse::BRef<Value> && dc);
Description. Move-analogues of previous constructor.

Object altering methods

void detach();
Description. Detach both pointers.

void detach_receive(const AVal<Value> & avc);
Description. Redirect activator and value pointer to clones of ownees of corresponding pointers of avc.

template<typename ... Args>
void detach_set_activate(Args && ... args);
Description. Redirect activator to envelope of true, and value pointer — to envelope of Value(args ...).

template<typename ... Args>
void detach_set_deactivate(Args && ... args);
Description. Redirect activator to envelope of false, and value pointer — to envelope of Value(args ...).

Object preserving, contents altering methods

`void activate() const;`
Description. Raise flag of activator.

`void deactivate() const;`
Description. Lower flag of activator.

`void receive(const AVal<Value> & avc) const;`
Description. Rewrite ownees of activator and value pointer with ownees of corresponding pointers of `avc`.

```
template<typename ... Args>
void set_activate(Args && ... args) const;
```
Description. Raise flag and record `Value(args ...)` to ownee of value pointer.

```
template<typename ... Args>
void set_deactivate(Args && ... args) const;
```
Description. Lower flag and record `Value(args ...)` to ownee of value pointer.

Object and contents preserving methods

`AData clone() const;`
Description. Return subcontext whose pointers own clones of ownees of coreesponding pointers of `*this`.

`void send(const AVal<Value> & avc) const;`
Description. Rewrite ownees of pointers of `avc` with ownees of corresponding pointers of `*this`.

Access methods

`bool is_active() const;`
Description. Return `true` $\Leftrightarrow$ value is active.

`bool is_inactive() const;`
Description. Return `true` $\Leftrightarrow$ value is inactive.

`Value & val() const;`
Description. Return value.

Fields

`Activator ac;`
Description. Activator (tricky pointer to boolean flag) of stored value.

`type_abuse::BRef<Value> vc;`
Description. Tricky pointer to stored value.

10.9 `class context::String;`

Location

`regexp/context/string.hpp`

Namespace of the definiton

`pm::regexp::context`

Description

Synonym for a tricky pointer to `std::string` which allows to shorten types of subcontext pointers a bit.

Definition

`typedef type_abuse::BRef<std::string> String`

10.10 Interface for basic regular expressions

`const Regexp anylet;`
Location. `regexp/basic/anylet.hpp`
Sort. Precise, 1 symbol.
Accepted strings. Accept any successfully read symbol.
Data change. Record actually read symbol to data.

`Regexp anyletbut(char bad_char);`
Location. `regexp/basic/anyletbut.hpp`
Sort. Precise, 1 symbol.
Accepted strings. Accept any successfully read symbol, except `bad_char`.
Data change. Record actually read symbol to data.

`Regexp anyletbutset(const std::set<char> & bad_chars);`
Location. `regexp/basic/anyletbutset.hpp`
Sort. Precise, 1 symbol.
Accepted strings. Accept any successfully read symbol, except symbols of set `bad_chars`.
Data change. Record actually read symbol to data.

`Regexp anyletbutset(std::set<char> && bad_chars);`
Location. `regexp/basic/anyletbutset.hpp`
Description. Move-analogue of previous function.

`Regexp anystr(size_t size);`
Location. `regexp/basic/anystr.hpp`
Sort. Precise, `size` symbols.
Accepted strings. Accept any successfully read string of size `size`.
Data change. Record actually read string to data.

`Regexp enclose(const std::string & opening_string, const std::string & closing_string);`
Location. `regexp/basic/enclose.hpp`
Sort. Imprecise.
Accepted strings. Accept the shortest string of the following form: `opening_string`, then any substring (including empty), then `closing_string`.
Data change. Record actually read string to data.
Comment. This expression is made basic, as incredibly often such is the structure of comments in programming languages.
For example, comments of `C++` can be described by the following two expressions:
1. `enclose(``/*'', ``*/'')`;
2. `enclose(``//'', ``\n'')`.

`Regexp enclose(const std::string & opening_string, std::string && closing_string);`

`Regexp enclose(std::string && opening_string, const std::string & closing_string);`

`Regexp enclose(std::string && opening_string, std::string && closing_string);`
Location. `regexp/basic/enclose.hpp`
Description. Move-analogues of previous function.

`const Regexp end;`
Location. `regexp/basic/end.hpp`
Sort. Precise, 1 symbol.
Accepted strings. Read one symbol, and immediately return in to the stream.
Accepted string is empty.
Success $\Leftrightarrow$ read operation failed.
Data change. Data remain unchanged.

`const Regexp fail;`
Location. `regexp/basic/fail.hpp`
Sort. Precise, 0 symbols.
Accepted strings. Always fails.

`Regexp let(char good_char);`
Location. `regexp/basic/let.hpp`
Sort. Precise, 1 symbol.
Accepted strings. Accept successfully read symbol, if it is equal to `good_char`.
Data change. `good_char` is recordeed to data.

`Regexp letset(const std::set<char> & good_chars);`
Location. `regexp/basic/letset.hpp`

Sort. Precise, 1 symbol.
Accepted strings. Accept any successfully read symbol, if it belongs to set `good_chars`.
Data change. Record actually read symbol to data.

`Regexp letset(std::set<char> && good_chars);`
Location. `regexp/basic/letset.hpp`
Description. Move-analogue of previous function.

`Regexp range(char bottom_char, char top_char);`
Location. `regexp/basic/range.hpp`
Sort. Precise, 1 symbol.
Accepted strings. Accept any successfully read symbol `c`, if `bottom_char <= c <= top_char`.
Data change. Record `c` to data.

`const Regexp skip;`
Location. `regexp/regexp.hpp`
Sort. Precise, 0 symbols.
Accepted strings. Always successfull, accepts empty string.
Data change. Data remains unchanged.

`Regexp str(const std::string & good_string);`
Location. `regexp/basic/str.hpp`
Sort. Precise, as many symbols as the length of `good_string`.
Accepted strings. Accept successfully read string `s` of size equal to size of `good_string`, if `s == good_string`.
Data change. Record `good_string` to data.

`Regexp str(std::string && good_string);`
Location. `regexp/basic/str.hpp`
Description. Move-analogue of previous function.

10.11 Interface for compositional operations

`Regexp operator |(const Regexp & r1, const Regexp & r2);`
Location. `regexp/composition/alt.hpp`
Accepted strings. Parse by `r1`.
If the parse is failed, parse by `r2`.
Success $\Leftrightarrow$ at least one of subexpressions returned success.
Accepted string is the string accepted by a successfully returned subexpression.
Data change. Data are changed by a successfull subexpression parse.

`Regexp operator |(const Regexp & r1, Regexp && r2);`

`Regexp operator |(Regexp && r1, const Regexp & r2);`

`Regexp operator |(Regexp && r1, Regexp && r2);`
Location. `regexp/composition/alt.hpp`
Description. Move-analogues of previous function.

`Regexp operator &(const Regexp & r1, const Regexp & r2);`
Location. `regexp/composition/concat.hpp`
Accepted strings. Parse by `r1`.
If the parse is successfull, parse the rest of the stream by `r2`.
Success $\Leftrightarrow$ both parses are sucessfull.
Accepted string is `s1 + s2`, where `s1` is acepted by `r1`, and `s2` is accepted by `r2`.
Data change. Data are changed consecutively by `r1` and `r2`.

`Regexp operator &(const Regexp & r1, Regexp && r2);`

`Regexp operator &(Regexp && r1, const Regexp & r2);`

`Regexp operator &(Regexp && r1, Regexp && r2);`
Location. `regexp/composition/concat.hpp`
Description. Move-analogues of previous function.

`Regexp operator !(const Regexp & r;`
Location. `regexp/composition/opt.hpp`
Accepted strings. Parse by `r`.
If the parse is successfull, return success and accept the string accepted by `r`.
Otherwise, return success and accept an empty string.
Data change. In case of failed subexpression parse, data remain unchanged.
Otherwise, data are changed by `r`.

`Regexp operator !(Regexp && r;`

Location. `regexp/composition/opt.hpp`
Description. Move-analogue of previous function.

`Regexp operator *(const Regexp & r);`
Location. `regexp/composition/star.hpp`
Accepted strings. Parse by `r` over and over until fail.
Accept the string `s1 + s2 + ... + sk`, where `k` is the number of successfull parses, and `si` is the string accepted by `i`-th parse.
Data change. Data are consecutively changed by `r` at each successull parse.

`Regexp operator *(Regexp && r);`
Location. `regexp/composition/star.hpp`
Description. Move-analogue of previous function.

10.12 Interface for functional operations

General description

The common operational part of regular expressions obtained with this interface is as follows. The only subexpression executes with a new context which has a different ownee of a tricky pointer for data. After the execution the initial data and the data obtained as the result of the execution are changed (in some cases - not changed).

`main` is a tricky pointer to initial data.

`sub` is a tricky pointer to data in the tricky pointer of the subexpression context.

"`sub = main`" means that these pointers are the same pointer.

A name of each funtional operation consists of two letters. The first letter means how `main` and `sub` are related. The second letter means how data are changed after execution of the subexpression.

"`a`" on the 2-nd place means application of `sub` to `main` with `type_abuse::Applier`.

"`c`" means that a new data is generated with `type_abuse::Creator`.

"`m`" means that data is modified with `type_abuse::Modifier`.

"`n`" on the 1-st place means that a data flag is lowered at the start of subexpression execution.

"`x`" means that data remains unchanged.

Operations

`Regexp ca(const type_abuse::Creator & cr, const Regexp & r, const type_abuse::Applier & ap);`
Location. `regexp/function/ca.hpp`
Data change. `sub = cr()`.
Ater execution `ap(main, sub)` is called.

`Regexp ca(const type_abuse::Creator & cr, const Regexp & r, type_abuse::Applier && ap);`

`Regexp ca(const type_abuse::Creator & cr, Regexp && r, const type_abuse::Applier & ap);`

`Regexp ca(const type_abuse::Creator & cr, Regexp && r, type_abuse::Applier && ap);`

`Regexp ca(type_abuse::Creator && cr, const Regexp & r, const type_abuse::Applier & ap);`

`Regexp ca(type_abuse::Creator && cr, const Regexp & r, type_abuse::Applier && ap);`

`Regexp ca(type_abuse::Creator && cr, Regexp && r, const type_abuse::Applier & ap);`

`Regexp ca(type_abuse::Creator && cr, Regexp && r, type_abuse::Applier && ap);`
Location. `regexp/function/ca.hpp`
Description. Move-analogues of previous function.

`Regexp cm(const type_abuse::Creator & cr, const Regexp & r, const type_abuse::Modifier & md);`
Location. `regexp/function/cm.hpp`
Data change. `sub = cr()`.
After execution `md(sub)` is called, and data of ownee of `main` is replaced by data of ownee of `sub`.

`Regexp cm(const type_abuse::Creator & cr, const Regexp & r, type_abuse::Modifier && md);`

`Regexp cm(const type_abuse::Creator & cr, Regexp && r, const type_abuse::Modifier & md);`

`Regexp cm(const type_abuse::Creator & cr, Regexp && r, type_abuse::Modifier && md);`

`Regexp cm(type_abuse::Creator && cr, const Regexp & r, const type_abuse::Modifier & md);`

`Regexp cm(type_abuse::Creator && cr, const Regexp & r, type_abuse::Modifier && md);`

`Regexp cm(type_abuse::Creator && cr, Regexp && r, const type_abuse::Modifier & md);`

`Regexp cm(type_abuse::Creator && cr, Regexp && r, type_abuse::Modifier && md);`
Location. `regexp/function/cm.hpp`
Description. Move-analogues of previous function.

`Regexp cx(const type_abuse::Creator & cr, const Regexp & r);`
Location. `regexp/function/cx.hpp`
Data change. `sub = cr()`.
After execution data of ownee of `main` is replaced by data of ownee of `sub`.

`Regexp cx(const type_abuse::Creator & cr, Regexp && r);`

`Regexp cx(type_abuse::Creator && cr, const Regexp & r);`

`Regexp cx(type_abuse::Creator && cr, Regexp && r);`
Location. `regexp/function/cx.hpp`
Description. Move-analogues of previous function.

`Regexp ma(const type_abuse::Modifier & md, const Regexp & r, const type_abuse::Applier & ap);`
Location. `regexp/function/ma.hpp`
Data change. `sub = md(main.clone())`.
After execution `ap(main, sub)` is called.

`Regexp ma(const type_abuse::Modifier & md, const Regexp & r, type_abuse::Applier && ap);`

`Regexp ma(const type_abuse::Modifier & md, Regexp && r, const type_abuse::Applier & ap);`

`Regexp ma(const type_abuse::Modifier & md, Regexp && r, type_abuse::Applier && ap);`

`Regexp ma(type_abuse::Modifier && md, const Regexp & r, const type_abuse::Applier & ap);`

`Regexp ma(type_abuse::Modifier && md, const Regexp & r, type_abuse::Applier && ap);`

`Regexp ma(type_abuse::Modifier && md, Regexp && r, const type_abuse::Applier & ap);`

`Regexp ma(type_abuse::Modifier && md, Regexp && r, type_abuse::Applier && ap);`
Location. `regexp/function/ma.hpp`
Description. Move-analogues of previous function.

`Regexp mm(const type_abuse::Modifier & md1, const Regexp & r, const type_abuse::Modifier & md2);`
Location. `regexp/function/mm.hpp`
Data change. `sub = md1(main.clone())`.
After execution `md2(sub)` is called, and data of ownee of `main` is replaced by data of ownee of `sub`.

`Regexp mm(const type_abuse::Modifier & md1, const Regexp & r, type_abuse::Modifier && md2);`

`Regexp mm(const type_abuse::Modifier & md1, Regexp && r, const type_abuse::Modifier & md2);`

`Regexp mm(const type_abuse::Modifier & md1, Regexp && r, type_abuse::Modifier && md2);`

`Regexp mm(type_abuse::Modifier && md1, const Regexp & r, const type_abuse::Modifier & md2);`

`Regexp mm(type_abuse::Modifier && md1, const Regexp & r, type_abuse::Modifier && md2);`

`Regexp mm(type_abuse::Modifier && md1, Regexp && r, const type_abuse::Modifier & md2);`

`Regexp mm(type_abuse::Modifier && md1, Regexp && r, type_abuse::Modifier && md2);`
Location. `regexp/function/mm.hpp`
Description. Move-analogues of previous function.

`Regexp mx(const type_abuse::Modifier & md, const Regexp & r);`
Location. `regexp/function/mx.hpp`
Data change. `sub = md(main.clone())`.
After exection data of ownee of `main` is replaced by data of ownee of `sub`.

`Regexp mx(const type_abuse::Modifier & md, Regexp && r);`

`Regexp mx(type_abuse::Modifier && md, const Regexp & r);`

`Regexp mx(type_abuse::Modifier && md, Regexp && r);`
Location. `regexp/function/mx.hpp`
Description. Move-analogues of previous function.

`Regexp nc(const Regexp & r, const type_abuse::Creator & cr);`
Location. `regexp/function/nc.hpp`
Data change. `sub` owns new empty block. Tricky pointer to data flag owns new envelope of `false`.
After execution data of ownee of `main` is replaced by data of ownee of `cr()`.

`Regexp nc(const Regexp & r, type_abuse::Creator && cr);`

`Regexp nc(Regexp && r, const type_abuse::Creator & cr);`

`Regexp nc(Regexp && r, type_abuse::Creator && cr);`
Location. `regexp/function/nc.hpp`
Description. Move-analogues of previous function.

`Regexp nm(const Regexp & r, const type_abuse::Modifier & md);`
Location. `regexp/function/nm.hpp`

Data change. `sub` owns new empty block. Tricky pointer to data flag owns new envelope of `false`. After execution `md(main)` is called.

`Regexp nm(const Regexp & r, type_abuse::Modifier && md);`

`Regexp nm(Regexp && r, const type_abuse::Modifier & md);`

`Regexp nm(Regexp && r, type_abuse::Modifier && md);`

Location. `regexp/function/nm.hpp`

Description. Move-analogues of previous function.

`Regexp nx(const Regexp & r);`

Location. `regexp/function/nx.hpp`

Data change. `sub` owns new empty block. Tricky pointer to data flag owns new envelope of `false`.

`Regexp nx(Regexp && r);`

Location. `regexp/function/nx.hpp`

Description. Move-analogue of previous function.

`Regexp xa(const Regexp & r, const type_abuse::Applier & ap);`

Location. `regexp/function/xa.hpp`

Data change. `sub = main.clone()`. After exection `ap(main, sub)` is called.

`Regexp xa(const Regexp & r, type_abuse::Applier && ap);`

`Regexp xa(Regexp && r, const type_abuse::Applier & ap);`

`Regexp xa(Regexp && r, type_abuse::Applier && ap);`

Location. `regexp/function/xa.hpp`

Description. Move-analogues of previous function.

`Regexp xm(const Regexp & r, const type_abuse::Modifier & md);`

Location. `regexp/function/xm.hpp`

Data change. `sub = main`. After execution `md(main)` is called.

`Regexp xm(const Regexp & r, type_abuse::Modifier && md);`

`Regexp xm(Regexp && r, const type_abuse::Modifier & md);`

`Regexp xm(Regexp && r, type_abuse::Modifier && md);`

Location. `regexp/function/xm.hpp`

Description. Move-analogues of previous function.

11 Interface of `dataset` module

All definitions are located in namespace

`pm::dataset`.

11.1 General description

The author recommends to consider this module as a part of `grammar` module which for some unknown reasons became independent. If someone is interested in this module itself, it is also ok.

Interface provided by this module can be split into the following parts:

1. Datasets. This part contains a class `Set` which generalizes tricky pointers to whatever data to collections of such pointers of a fixed given size.
2. Dataset functions. This part contains:
   - `FBase` class, which is a base for classes, objects of which mean "it can be applied to a dataset to perform some actions on it";
   - `Function` class, which is derived from `FBase`, and stores a tricky pointer of type `type_abuse::CRef<FBase>`, — this class provide means for a nontemplate uniform management of objects of classes derived from `FBase` (i.e. dataset functions);
   - interface for creation of some ready-to-use sorts of dataset functions; to use all these sorts at once, include the header `dataset/functions.hpp`.
3. Syntax. Interface of this part is placed in a subspace `syntax` of the namespace of the module. This interface provides handy means for creation of datasets and dataset functions based on
   - syntactic descriptions of dataset and dataset functions in which every offset to a pointer of a dataset is described with an assigned name (a string), and
   - interface for translation of syntactic descriptions to datasets and dataset functions.

11.1.1 Translation principles

Translation interface is simply several global functions — translation functions — with a common name `syntax::interpret`. There exist two sorts of translation:

1. Translation of a dataset description to a dataset.
2. Translation of a dataset function description in context of a dataset description to a dataset function.

A dataset description is an object of class `syntax::Set`. A dataset function description is an object of class `syntax::Function`.

When `interpret` function is called, all offset names should be designed (a distinct name means a distince offset) and added to a dataset description with special methods. A name can be added to a dataset description explicitly, or via methods which add all names contained in a dataset function. A name is an arbitrary string.

A dataset is obtained as a result of a translation function call with the only argument — a dataset description. A dataset function is obtained as a resuly of a translation function call with two arguments: a dataset function description, and a dataset description which describes a mapping of names into offsets in a dataset vector.

A user should not care about internals of the class `syntax::Function`, as the only important things about objects of this class are how to obtain them, and how to translate them into dataset functions. Due to this fact, `syntax::Function` class is not described in this manual.

11.2 `class FBase;`

Location

`dataset/fbase.hpp`

Description

Base class of dataset functions.

Default constructors and assignments, destructor

All are present, standard semantics.
Destructor is virtual.

Methods

`virtual void apply(const Set & ds) const;`
Description. Immediately returns without any effect on `ds`.
The main intention of this method is to be overloaded in derived classes.

11.3 `class Function : public FBase;`

Location

`dataset/function.hpp`

Description

Class containing a tricky pointer to an object of type derived from `FBase`.
An overloaded method `apply` calls the same method of a stored object with the same argument.
The main intention of this class is to provide means for nontemplate unified management of dataset functions.

Default constructors and assignments, destructor

All are present, standard semantics.

Other constructors

`Function(const type_abuse::CRef<FBase> & pf);`
Description. Direct pointer to ownee of `pf`.
Requirements. If `this->apply` is to be called, then ownee of `pf` is nonempty.

`Function(type_abuse::CRef<FBase> && pf);`
Description. Move-analogue of previous constructor.

Methods

`void apply(const Set & ds) const override;`
Description. Apply stored object to `ds`.

11.4 `class Set;`

Location

`dataset/set.hpp`

Description

Class of datasets.
A dataset is a vector of tricky pointers to whatever data, wrapped in methods which, on the one hand, restrict vector management (e.g. deny size change), and on the other hand, provide some handy means of pointer management.

Default constructors and assignments, destructor

`Set()` is deleted.
The rest are present, standard semantics up to default actions on tricky pointers.

Other constructors

`Set(std::initializer_list<type_abuse::DRef> l);`
Description. Initialize vector of the same size as `l`.
Direct pointers to ownees of elements of `l`.

`Set(const std::vector<type_abuse::DRef> & v);`
Description. Initialize vector of the same size as `v`.
Direct pointers to ownees of elements of `v`.

`Set(std::vector<type_abuse::DRef> && v);`
Description. Move-analogue of previous constructor.

`Set(size_t size);`
Description. Initialize vector of size `size`.
Direct each pointer to its own new empty block.

Other operators

`type_abuse::DRef & operator [](size_t i);`
Description. Return pointer to `i`-th data.
Requirements. Vector has size greater or equal to `i+1`.

`const type_abuse::DRef & operator [](size_t i) const;`
Description. Const-analogue of previous operator.

Object altering methods

`void detach();`
Description. Detach each pointer.

`void detach_receive(const Set & set);`
Description. Redirect each pointer of `*this` to clone of ownee of pointer of `set` with the same offset.
Requirements. Size of vector of `set` is greater or equal to size of vector of `*this`.

`void detach_unset();`
Description. Redirect each pointer to its own new empty block.

Object preserving, contents altering methods

`void receive(const Set & set) const;`
Description. Rewrite ownee of each pointer of `*this` with ownee of pointer of `set` with the same offset.
Requirements. Size of vector of `set` is greater or equal to size of vector of `*this`.

`void unset() const;`
Description. Make ownee of each pointer empty.

Object and contents preserving methods

`Set clone() const;`
Description. Return dataset of the same size as `*this`, such that each pointer of the result owns clone of ownee of pointer of `*this` with the same offset.

`void send(const Set & set) const;`
Description. Rewrite first ownees of pointers of `set` (as much as size of `*this`) with ownees of pointers of `*this` with the same offsets.
Requirements. Size of vector of `set` is greater or equal to size of vector of `*this`.

Access methods, except iteration

`type_abuse::DRef & at(size_t i);`
Description. Return `i`-th pointer.
Requirements. Size of vector is greater or equal to `i+1`.

`const type_abuse::DRef & at(size_t i) const;`
Description. Const-analogue of previous method.

`size_t size() const;`
Description. Return size of vector.

Methods for iteration

`std::vector<type_abuse::DRef>::iterator begin();`
Description. Return iterator to beginning of vector.

`std::vector<type_abuse::DRef>::const_iterator begin() const;`
Description. Const-analogue of previous method.

`std::vector<type_abuse::DRef>::iterator end();`
Description. Return iterator to end of vector.

`std::vector<type_abuse::DRef>::const_iterator end() const;`
Description. Const-analogue of previous method.

11.5 Interface for creation of dataset functions

General description

Further descriptions are simply actions performed over a dataset `ds` when `apply(ds)` is called for a returned function.

Interface

`Function applier(size_t t, size_t s, const type_abuse::Applier & f);`
Location. `dataset/functions/applier.hpp`
Operation. `f(ds[t], ds[s])`.

`Function applier(size_t t, size_t s, type_abuse::Applier && f);`
Location. `dataset/functions/applier.hpp`
Description. Move-analogue of previous function.

`Function cloner(size_t t, size_t s);`
Location. `dataset/functions/cloner.hpp`
Operation. Rewrite ownee of `ds[t]` with ownee `ds[s]`.

`Function composition(const std::list<Function> & funcs);`
Location. `dataset/functions/composition.hpp`
Operation. Consequently call `apply(ds)` for elements of `funcs`.

`Function composition(std::list<Function> && funcs);`
Location. `dataset/functions/composition.hpp`
Description. Move-analogue of previous function.

`Function creator(size_t t, const type_abuse::Creator & f);`
Location. `dataset/functions/creator.hpp`
Operation. Rewrite ownee of `ds[t]` with ownee of `f()`.

`Function creator(size_t t, type_abuse::Creator && f);`
Location. `dataset/functions/creator.hpp`
Description. Move-analogue of previous function.

`Function flusher(size_t t, const type_abuse::DRef & d);`
Location. `dataset/functions/flusher.hpp`
Operation. Rewrite ownee of `ds[t]` with ownee of `d`.

`Function applier(size_t t, type_abuse::DRef && d);`
Location. `dataset/functions/flusher.hpp`
Description. Move-analogue of previous function.

`Function modifier(size_t t, const type_abuse::Modifier & f);`
Location. `dataset/functions/modifier.hpp`
Operation. `f(ds[t])`.

`Function modifier(size_t t, type_abuse::Modifier && f);`
Location. `dataset/functions/modifier.hpp`
Description. Move-analogue of previous function.

11.6 `class syntax::Set;`

Location

`dataset/syntax/set.hpp`

Namespace of the definiton

`pm::dataset::syntax`

Description

Class of syntactic descriptions of datasets.
This description is one of the arguments of translation functions which allow to obtain a dataset, or a dataset function.
At each moment of lifetime, a description contains a set of names (strings).

Default constructors and assignments, destructor

All are present, standard semantics.
`Set()` initializes empty set of names.

Object altering methods

`size_t add(const std::string & name);`
Description. Add `name` to set of names, if not present.
In any case return offset of pointer to data to which `name` will be translated until set of names is cleaned.

`size_t add(std::string && name);`
Description. Move-analogue of previous method.

`void add(const Function & fun);`
Description. Add all names used in `fun` to set of names.

`void add(Function && fun);`
Description. Move-analogue of previous method.

`void clear();`
Description. Clean set of names.

Access methods

`bool has_name(const std::string & name) const;`
Description. Return `true` $\Leftrightarrow$ name `name` is contained in set of names.

`size_t index_of(const std::string & name) const;`
Description. Return offset of pointer to data to which `name` will be translated until set of names is cleaned.
Requirements. Name `name` is contained in set of names.

`const utils::Numeration<std::string> & names() const;`
Description. Return numeration which describes offsets of pointers to which names of current set will be translated until set of names is cleaned.

`size_t size() const;`
Description. Return size of set of contained names.

11.7 Translation interface

Location

`dataset/syntax/interpreter.hpp`

Namespace of the definiton

`pm::dataset::syntax`

Interface

`dataset::Set interpret(const Set & set);`
Description. Return translation result of description `set` to dataset.
Size of dataset is equal to number of distinct names contained in `set`.
Each name corresponds to disctinct offset in vector.
Each tricky pointer of the result owns its own new empty block.

`dataset::Function interpret(const Function & fun, const Set & set);`
Description. Return dataset function to which `fun` is translated in context of dataset description `set` according to general description of translation.
Requirements. Each name used in `fun` is contained in `set`.

`dataset::Function interpret(Function && fun, const Set & set);`
Description. Move-analogue of previous function.

11.8 Interface for creation of syntactic descriptions of dataset functions

Location

`dataset/syntax/function.hpp`

Namespace of the definiton

`pm::dataset::syntax`

General description

This interface allows to obtain objects of type `Function` which are used as arguments of translation functions for obtaining dataset functions. Interface description says to which dataset function the description is translated in context of a dataset description. To say it shorter, the following notation is used: `<name>` is an offset of a tricky pointer to whatever data to which `name` is translated in context of a given dataset description.

Interface

`Function trivial_function;`
Correspondence. Trivial function `dataset::Function()`.

`Function applier(const std::string & target, const std::string & source, const type_abuse::Applier & fun);`
Correspondence. `dataset::applier(<target>, <source>, fun)`

`Function applier(const std::string & target, const std::string & source, type_abuse::Applier && fun);`

`Function applier(const std::string & target, std::string && source, const type_abuse::Applier & fun);`

`Function applier(const std::string & target, std::string && source, type_abuse::Applier && fun);`

`Function applier(std::string && target, const std::string & source, const type_abuse::Applier & fun);`

`Function applier(std::string && target, const std::string & source, type_abuse::Applier && fun);`

`Function applier(std::string && target, std::string && source, const type_abuse::Applier & fun);`

`Function applier(std::string && target, std::string && source, type_abuse::Applier && fun);`
Description. Move-analogues of previous function.

`Function cloner(const std::string & target, const std::string & source);`
Correspondence. `dataset::cloner(<target>, <source>)`.

`Function cloner(const std::string & target, std::string && source);`

`Function cloner(std::string && target, const std::string & source);`

`Function cloner(std::string && target, std::string && source);`
Description. Move-analogues of previous function.

`Function composition(const std::list<Function> & funcs);`
Correspondence. `dataset::composition(<funcs>)`, where `<funcs>` is list of translation results for elements of `funcs`.

`Function composition(std::list<Function> && funcs);`
Description. Move-analogue of previous function.

`Function creator(const std::string & target, const type_abuse::Creator & fun);`
Correspondence. `dataset::creator(<target>, fun)`.

`Function creator(const std::string & target, type_abuse::Creator && fun);`

`Function creator(std::string && target, const type_abuse::Creator & fun);`

`Function creator(std::string && target, type_abuse::Creator && fun);`
Description. Move-analogues of previous function.

`Function flusher(const std::string & target, const type_abuse::DRef & data);`
Correspondence. `dataset::flusher(<target>, data)`.

`Function flusher(const std::string & target, type_abuse::DRef && data);`

`Function flusher(std::string && target, const type_abuse::DRef & data);`

`Function flusher(std::string && target, type_abuse::DRef && data);`
Description. Move-analogues of previous function.

`Function modifier(const std::string & target, const type_abuse::Modifier & fun);`
Correspondence. `dataset::modifier(<target>, fun)`.

`Function modifier(const std::string & target, type_abuse::Modifier && fun);`

`Function modifier(std::string && target, const type_abuse::Modifier & fun);`

`Function modifier(std::string && target, type_abuse::Modifier && fun);`
Description. Move-analogues of previous function.

12 Interface of `grammar` module

All definitions are located in namespace

`pm::grammar`.

12.1 General description

12.1.1 What is grammar

The notion of grammars is well-known — at least to some programmers who tried to parse strings and streams. This notion is trickier than the notion of regular expressions, and is used to parse heavy well-structured texts, such as (but not limited to) program code. The author tried to write this manual as understandable as possible, even for those who had no needs to use grammars (but know what a mathematics is). The initial explanation point is a grammar in mathematics.

First of all, note that grammars implemented in this modules are something like context-free grammars. Due to this fact all definitions use the term "grammars" implicitly assuming context-free grammars.

A grammar G over a finite alphabet $\mathcal{A}$ (a finite set of symbols; an alphabet of terminals) consists of:

1. A finite set of nonterminals $\mathcal{N}$.
2. An initial nonterminal S, $S \in \mathcal{N}$.
3. A finite set of rules $\mathcal{R}$. Each rule has the form $A \to BODY$, where A is a left-hand side of the rule — a nonterminal — and $BODY$ is a right-hand side — a finite sequence of terminals and nonterminals.

A meaning (semantics) of a grammar G, as for regular expressions, is a language $L(G)$, which is a set of strings accepted by the grammar. This language may be defined as follows. Take a string "S", and transform it step by step, obtaining a string of terminals and nonterminals at each step. A transformation step looks as follows:

- take one (any) nonterminal in a current string, and one (any) rule whose left-hand side is this nonterminal;
- replace this nonterminal in the string with the right-hand side of the rule.

A string is accepted by the grammar $\Leftrightarrow$ there exists a sequence of described transformation steps of S which ends with the string.

A typical example of a nontrivial grammar is a grammar of balanced brackets. In a simple case it contains one nonterminal S and two rules: $S \to (S)S$ and $S \to \lambda$ — where λ is an empty string. It can be easily shown that the language of this grammar is the set of all strings of balanced opening and closing brackets.

Grammars in programming, just like regural expressions in programming, usually have wider syntax and narrower semantics than those of mathematical grammars. On the one hand, new structures and transformation details are added. On the other hand, not all strings accepted in mathematical sense are accepted in programming sense.

The most common use of grammars in programming is to solve the following problem: given a string, to check it it accepted by a grammar, and if yes, then perform some actions depending on the certain structure of the string. To solve this problem, some internal structures are created (sometimes based on strings over terminals and nonterminals), and symbold of the given string are read, which is called a parse of the string. Due to mechanics of standard input streams, and of streams in general, the most programming kind of parse is a left-to-right parse, which means that symbols of the string are observed (read) one by one from the leftmost (the first one) to the rightmost (the last one). We use a left-to-right parse with rollback, which means that we allow to unread symbols of the string (put them back to the stream).

This module uses a certain parsing procedure. For those who know, it is a recursive descent parse with rollback. It works as follows. All the rules of the grammar are ordered. A procedure "parse by an initial nonterminal" is called.

For each nonterminal there is a procedure "parse by this nonterminal", which works as follows:

1. Call a procedure "parse by first rule such that its left-hand side is a current nonterminal".
2. If a call is successfully finished, then successfully finish the current procedure, otherwise call a procedure "parse by second rule susch that its left-hand side is a current nonterminal".
3. ...
4. If all the rules whose left-hand side is a current nonterminal are failed, then the current procedure is failed (unsuccessfully finised).

A procedure "parse by a rule" works as follows. Process a right-hand side of the rule symbol by symbol, and do the following for each symbol:

1. If the current symbol is a terminal, then try to read the next symbol of the string. If this symbol is equal to the terminal, then continue, otherwise the procedure is failed.
2. If the current symbol is a nonterminal, then call a procedure "parse by this terminal" (for a substring starting with the next unread symbol). If the call is successfully finished, then continue, otherwise the procedure is failed.
3. If the last call is successfully finished, then the procedure is successfully finished.
4. If the procedure is failed, unread all symbols as if the procedure was never called.

It can be easily seen that the described parse accepts less words than in mathematical sense. Nevertheless, such kind of parse is also nontrivial and highly useful.

The next question is how to perform useful actions during the parse depending on the structure of the parsed string, except reading of its symbols and calling subparses. For instance,[17] we may want not only to accept a balanced bracket-only string, but to count the number of bracket pairs contained on the string. The question is answered by each grammar developer in his own way. The author gives the following answer. The parse is being split into primitive grammar actions, such as "parse by a nonterminal", "parse by a rule", and "read nonterminal". Each grammar action can be wrapped with data actions — for instance, the grammar for balanced brackets can be modified in the following way. Take two nonterminals (an initial S, and an additional Q). Make three rules instead of two: $S \to Q$, $Q \to (Q)Q$, $Q \to \lambda$. Perform an action "initialize the internal variable with zero" before "parse with S". Perform an action "increase the internal variable", if the procedure "parse with $Q \to (Q)Q$" is successfully finished. The number of pairs of brackets is stored into the internal variable after the successfull parse.

The last remark on grammars is: commonly, strings parsed by grammars contain so-called "blank symbols". When a string is parsed, blank symbols are omitted, i.e. read without any effect to grammar actions and data actions. For instance, blank symbols of `C++` language are `' '`, `'
t'`, and `'
n'`.

12.1.2 Grammars in `grammar` module

`grammar` provides two ways of grammar creation: direct, and syntactical.

Direct way is to create a "pure" grammar (class `Grammar`) which has unnamed components only. To refer to these components, unique indexes of these components are used (which are offsets in vector-alike structures).

Syntactical way of grammar creation (with a class `syntax::Grammar`) is closer to a mathematical grammar description: some components of a grammar a linked to unique names (strings), and may be referred to by these names. After syntactical description of a grammar is created, a translator should be used (class `syntax::Interpreter`) to automatically generate a pure grammar from its syntactical description. Comparing to the direct way, all syntactical performance overhead is located between creation of an object of type `syntax::Grammar`, and return of a pure grammar after a translation. After a pure grammar is obtained, its syntactical description may be destroyed.

`Grammar` class is inherited from a class `regexp::Literal`.[18] It means that grammars can parse strings, standard input streams, and reversable streams with and without contexts of type `regexp::Context` (regular contexts). The same parse conventions hold for grammars as for regular expressions, except one: if a data activity flag is raised, and a parse is failed, then data may still be changed. This convention is broken for a reason: grammars may manage large amounts of data, and to keep data unchanged may mean to duplicate large amounts of data, which is highly ineffective. Instead a user has means to "clean after himself": user-defined data functions which are called after failed subparses.

[17] Do not swear a lot because of such a dumb example.

[18] So, a grammar is not a special case of a regular expression, but a special case of a literal.

Additionally to `match` methods, `Grammar` class has `gmatch` methods (grammar parse methods) analogous to `match`, but taking a special grammar context (class `Context`) instead of a regular one. The difference between these contexts is that instead of a pointer to whatever data (`type_abuse::DRef`), a gramaar context contains a dataset (`dataset::Set`). All non-broken conventions are applied to `gmatch` methods: the data-related conventions apply to each data of the dataset (which is being activated as a whole with a single flag).

`match` and `gmatch` parses are connected with the notion of a main data. When `match_full` is called,

- a new grammar context is created with copied pointers to an accepted string, and to a string flag;
- in the new context a dataset is initialized of the size defined by the grammar;
- a pointer to main data in the dataset is copied from a data pointer of a regular context;
- an activity flag block is rewrited by an activity flag block of the regular context (but no pointers are redirected here);
- `gmatch_full` is called with the same stream and the new context.

12.1.3 Pure grammars

The most important primitive component of a grammar (of the module `grammar`) is a pure action. A pure action defines the main parse control flow. Currently the following pure actions are implemented:

1. An alternative (`alt`). It contains a vector of indexes of grammar nodes (a node is defined further). To execute this action is to execute the nodes one by one until the first successfull subexecution. An alternative is executed successfully $\Leftrightarrow$ there was a successfull subexecution. This action is basically a procedure "parse by a nonterminal" with rules replaced by nodes.
2. A rule (`rule`). It contains a vector of indexes of grammar nodes. To execute this action is to execute the nodes one by one. The next node continues reading symbols when the previous one stopped. A rule is executed successfully $\Leftrightarrow$ all subexecutions were successfull. This action is basically a procedure "parse by a rule" with terminals and nonterminals replaced by nodes.
3. A terminal (`re`). To execute this action is to parse a stream with a blank expression of the grammar (explained furhter), and then — by a regular expression contained in the action. This action is a generalization of the notion of terminal in mathematical grammars: to read a symbol a is to execute an action `regexp::let(`a`)`.
4. A forwarding (`next`). To execute this action is to execute a node whose index is contained in the action.
5. A trivial action. When executed, this action immediately returns success.

A sequential execution of pure action components until the first success basically means the a recursive descent parse with rollback is implemented.

A grammar action is a pure action which can be optionally extended with a related data index (in a dataset of a grammar context). Currently the index is ignored for all actions excep a nonterminal. A terminal without a related data index means that its regular expression is executed with a new empty data, and a new lowered data flag. A terminal with a related data index means that its regular expression is executed for a context with a data pointer copied from an element of the dataset defined by the index, and a new raised data flag. Ponters to string and string flag are copied from the grammar context in any case.

A grammar node consists of 4 components: a grammar action, and 3 functions over dataset (`dataset::Function`), namely a preparation function, a finalization function, and a cleaning function. To execute a node is to do the following. The preparation function is called (over a dataset of a current context). Then the action is executed. If the subexecution is successfull, then the finalization function is called, and the execution is successfully finished. If the subexecution is failed, then the cleaning function is called, and the execution is failed.

When talking about grammars in programming, usually the notion of a blank symbol emerges sooner or later. A blank symbol is a symbol to be ignored during read, and is used to format a text, and to separate different text structures. `grammar` module generalizes this notion to a blank regular expression. This is an expression which is executed over and over until the first fail right before a terminal. Its execution does not have any influence on a successfullness of a parse, as well as on generation of data. Blank expressions, however, are collected in an accepted string, and are returned to the stream when other symbols are returned. For instance, blank symbols of `C++` are `' '`, `'
t'`, `'
n'`, and the corresponding blank expression is `regexp::letset({' ', '
t', '
n'})`.

A grammar (of type `Grammar`) consists of an indexed collection of nodes (each node has its unique index), an initial node index, a blank expression, a dataset size, and — optionally — a main data index. If `match` is called with raised data flag, and a grammar has no main data index, then the parse immediately fails.

12.1.4 Syntactic descriptions of grammars

An interface for management of syntactic mechanisms for generation of pure grammars is located in a subspace `syntax` of a namespace of the module. This interface is designed to be used as follows:

1. Create and fill a syntactic description of a grammar (an object of type `syntax::Grammar`).

2. Create a translator (of typ `syntax::Interpreter`) and obtain a pure grammar by its syntactic description as the result of a call to the method `interpret` which takes the description as an argument.

A syntactic grammar description is filled with syntactic descriptions of components of a pure grammar. These descriptions do not use indexes for nodes and data. Instead, the following is allowed:

1. To assign names to nodes and data: namespaces for nodes and data are independent, and inside a namespace each object (a node, or a data; to be more precise — intended indexes of a node, or a data) is assigned to a single name, and distinct names are assigned to distinct objects.
2. To define nested descriptions to be translated into sets of nodes (instead of single nodes).
3. To use names instead of indexes when indexes should be used.

`dataset` module contains detailed description of how syntactic part of datasets operates. No explicit syntactic description of a dataset (`dataset::syntax::Set`) is needed when a grammar description is constructed — a dataset description is generated automatically with all names for data used during grammar description management.

If no initial node name is set, then a grammar description is translated to a grammar with a trivial initial node: it has a trivial pure action, and no other components.

A grammar description can be filled not only by descriptions of pure actions, but also by descriptions of nonterminals (`nonterminal`). A nonterminal contains a vector of vectors of pure action descriptions. The internal vectors are assumed to be contained in unnamed rules. The external vector unites these rules in an alternative. When a nonterminal is added, its name is assigned to the external alternative.

If a name used in a grammar description was not assigned to any node description, a trivial node is assumed.

A connection between a name and a node description may be deleted. Each method for node addition deleted a current description of a node, and replace it with a new description.

If no blank expression was added to a grammar description, then the expression `regexp::fail` is assumed, which means no blank symbols.

12.2 Interface for creation of pure actions

Location

`grammar/action.hpp`

Interface

`Action trivial_action;`
Description. Trivial action.

`Action alt(const std::vector<size_t> & nodes);`
Description. Return alternative containing copy of `nodes`.

`Action alt(std::vector<size_t> && nodes);`
Description. Move-analogue of previous function.

`Action next(size_t node);`
Description. Return forwarding containing `node`.

`Action re(const regexp::Regexp & re);`
Description. Return terminal containing copy of `re`.

`Action re(regexp::Regexp && re);`
Description. Move-analogue of previous function.

`Action rule(const std::vector<size_t> & nodes);`
Description. Return rule containing copy of `nodes`.

`Action rule(std::vector<size_t> && nodes);`
Description. Move-analogue of previous function.

12.3 `class Action;`

Location

`grammar/action.hpp`

Description

Class of grammar actions.

Default constructors and assignments, destructor

All are present, standard semantics up to default actions for tricky pointers of regular expressions.
`Action()` initializes trivial pure action with no related data.

Types

`enum T {TRIVIAL, ALTERNATIVE, NEXT, REGEXP, RULE};`
Description. `TRIVIAL` means trivial action.
`ALTERNATIVE` means alternative.
`NEXT` means forwarding.
`REGEXP` means terminal.
`RULE` means rule.

Altering methods

`void set_alt(const std::vector<size_t> & nodes);`
Description. Reset pure action to alternative containing vector `nodes`.

`void set_alt(std::vector<size_t> && nodes);`
Description. Move-analogue of previous method.

`void set_data(size_t data);`
Description. Set related data index to `data`.

`void set_next(size_t node);`
Description. Reset pure action to forwarding containing index `next(node)`.

`void set_re(const regexp::Regexp & re);`
Description. Reset pure action to terminal containing regular expression `re`.

`void set_re(regexp::Regexp && re);`
Description. Move-analogue of previous method.

`void set_rule(const std::vector<size_t> & nodes);`
Description. Reset pure action to rule containing vector `nodes`.

`void set_rule(std::vector<size_t> && nodes);`
Description. Move-analogue of previous method.

`void unset();`
Description. Reset pure action to trivial. Unset related data index.

`void unset_control();`
Description. Reset pure action to trivial.

`void unset_data();`
Description. Unset related data index.

Type access methods

`T type() const;`
Description. Return type of current pure action.

Presence-checking methods

`bool has_data() const;`
Description. Return `true` $\Leftrightarrow$ any data index is set.

`bool has_next() const;`
Description. Return `true` $\Leftrightarrow$ current pure action contains index of next node.

`bool has_nodes() const;`
Description. Return `true` $\Leftrightarrow$ current pure action contains vector of indexes nodes.

`bool has_re() const;`
Description. Return `true` $\Leftrightarrow$ current pure action contains regular expression.

Access methods

`size_t & data();`
Description. Return related data index.
Requirements. Related data index is set.

`const size_t & data() const;`
Description. Const-analogue of previous method.

`size_t & next();`
Description. Return index of next node contained in current pure action.
Requirements. Current pure action contains index of next node.

`const size_t & next() const;`
Description. Const-analogue of previous method.

`std::vector<size_t> & nodes();`
Description. Return vector of nodes contained in current pure action.
Requirements. Current pure action contains vector of nodes.

`const std::vector<size_t> & nodes() const;`
Description. Const-analogue of previous method.

`regexp::Regexp & re();`
Description. Return regular expression contained in current pure action.
Requirements. Current pure action contains regular expression.

`const regexp::Regexp & re() const;`
Description. Const-analogue of previous method.

12.4 `class Context;`

Location

`grammar/context.hpp`

Description

Class of grammar contexts.
It contains two fields (subcontexts):
1. An activational subcontext `asc` for an accepted string.
2. An activational subcontext `adc` for a dataset.

Up to internal structure of the subcontexts, a context contains 4 components:
1. A tricky pointer to an accepted string.
2. A string activator (a tricky pointer to a boolean flag which activated a string).
3. A dataset.
4. A dataset activator.

Default constructors and assignments, destructor

`Context()` is deleted.
The rest are present, standard semantics up to default actions for tricky pointers of regular expressions.

Other constructors

`Context(size_t size);`
Description. Initialize `*this` with a pointer to: a new empty string; a dataset of size `size`, each pointer of the set owns its own new empty block; lowered flags in new blocks.

`Context(const regexp::context::AString & asc, const context::ADataSet & adc);`
Description. Initialize `*this` with dataset of the same size as in `adc`, and direct all pointers (to string, to data, to flags) to ownees of corresponding pointers of arguments.

`Context(const regexp::context::AString & asc, context::ADataSet && adc);`

`Context(regexp::context::AString && asc, const context::ADataSet & adc);`

`Context(regexp::context::AString && asc, context::ADataSet && adc);`
Description. Move-analogues of previous constructor.

Altering methods

`void detach();`
Description. Detach all pointers (to string, to data, to flags).

`void detach_receive(const Context & sdc);`
Description. Redirect pointers to string and to flags of `*this` to clones of ownees of corresponding pointers of `sdc`. Call `detach_receive` for dataset of `*this` with argument — dataset of `sdc`.
Requirements. Inherited from `detach_recive` for dataset: size of dataset of `sdc` is greater or equal to size of dataset of `*this`.

Object-preserving, data-altering methods

`void receive(const Context & sdc) const;`

Description. Rewrite ownees of pointers to string and flags of `*this` with ownees of corresponding pointers of `sdc`. Call `receive` for dataset of `*this` with argument — dataset of `sdc`.

Requirements. Inherited from `receive` for dataset: size of dataset of `sdc` is greater or equal to size of dataset of `*this`.

Object- and data-preserving methods

`Context clone() const;`

Description. Return context of the same size of dataset as in `*this`, such that all pointers of the result are directed to clones of ownees of corresponding pointers of `*this`.

`void send(const Context & sdc) const;`

Description. Rewrite ownees of pointers to string and flags of `sdc` with ownees of corresponding blocks of `*this`. Call `send` for dataset of `*this` with argument — dataset of `sdc`.

Requirements. Inherited from `send` for dataset: size of dataset of `sdc` is greater or equal to size of dataset of `*this`.

Access methods

`type_abuse::DRef & data(size_t i);`

Description. Return pointer to `i`-th data in dataset.

Requirements. Size of dataset is greater of equal to `i+1`.

`const type_abuse::DRef & data(size_t i) const;`

Description. Const-analogue of previous method.

`size_t data_size() const;`

Description. Return size of dataset.

```
template<typename Value>
Value & data_value(size_t i) const;
```

Description. Return `i`-th data of dataset, assumed to be data of type `Value`.

Requirements. Size of dataset is greater or equal to `i+1`.
Ownee of pointer to `i`-th data is nonempty and stores data of type `Value`.

`bool is_active() const;`

Description. Return `true` ⇔ at least one of flags is raised.

`bool is_data_empty(size_t i) const;`

Description. Return `true` ⇔ ownee of pointer to `i`-th data of dataset os empty.

`bool is_data_nonempty(size_t i) const;`

Description. Return `true` ⇔ ownee of pointer to `i`-th data of dataset os nonempty.

`bool is_inactive() const;`

Description. Return `true` ⇔ both flags are lowered.

`std::string & string() const;`

Description. Return accepted string.

Fields

`regexp::context::AString asc;`

Description. Subcontext containing pointer to accepted string, and activator — pointer to string activity flag.

`context::ADataSet adc;`

Description. Subcontext containing dataset, and activator — pointer to dataset activity flag.

12.5 `class Grammar : public regexp::Literal;`

Location

`grammar/grammar.hpp`

Description

Class of grammars (to clarify — of pure grammars).
Methods of these class does not allow to nicely set dataset functions to nodes. There are two ways to do it. The first one is to prepare a node in which functions are set, and add it to a grammar with special methods. The second one is to obtain a reference to a node and set dataset functions with methods of the node.
To add a pure action is to add a node in which nothing except this pure action is present. When a node is added, an index of the node is returned which is not equal to any previous index.
Except all methods described further, all methods of `regexp::Literal` are inherited except an overloaded full parse method `match_full`. A full parse methos is overloaded in a way presented in general description.

Default constructors and assignments, destructor

All are present, standard semantics up to default actions for tricky pointers of regular expressions.
`Grammar()` initializes grammar with one trivial initial node, no blank expression (i.e. a blank expression `regexp::fail`), a zero-sized dataset, and an absent main data index.

Methods for addition of new actions

`size_t push_alt(const std::vector<size_t> & nodes);`
Description. Add alternative containing `nodes`.
Return index of added node.

`size_t push_alt(std::vector<size_t> && nodes);`
Description. Move-analogue of previous method.

`size_t push_next(size_t node);`
Description. Add forwarding containing `node`.
Return index of added node.

`size_t push_node();`
Description. Add trivial action.
Return index of added node.

`size_t push_node(const Node & node);`
Description. Add copy of node `node`.
Return index of added node.

`size_t push_node(Node && node);`
Description. Move-analogue of previous method.

`size_t push_re(const regexp::Regexp & r);`
Description. Add terminal containing `r`.
Return index of added node.

`size_t push_re(regexp::Regexp && re);`
Description. Move-analogue of previous method.

`size_t push_rule(const std::vector<size_t> & nodes);`
Description. Add rule containing `nodes`.
Return index of added node.

`size_t push_rule(std::vector<size_t> && nodes);`
Description. Move-analogue of previous method.

Altering methods except addition of new methods

`void set_data_size(size_t size);`
Description. Set dataset size to `size`.

`void set_main_data_index(size_t i);`
Description. Set main data index to `i`.

`void set_main_index(size_t i);`
Description. Set initial node index to `i`.

`void set_nodes(const std::vector<Node> & nodes);`
Description. Replace node vector of grammar with `nodes`.

`void set_nodes(std::vector<Node> && nodes);`
Description. Move-analogue of previous method.

`void set_skip(const regexp::Regexp & re);`
Description. Set blank expression to `re`.

`void set_skip(regexp::Regexp && re);`

Description. Move-analogue of previous method.

`void unset_main_data_index();`
Description. Delete main data index.

Parse methods

`bool gmatch(stream::Stream & s, Context & gcontext = default_context) const;`
Description. Call `gmatch_full` with same arguments, and return its result.

`bool gmatch(std::istream & s, Context & gcontext = default_context) const;`
Description. Call `gmatch` with `stream::Stream(s)` and same context, and return result of call.

`bool gmatch(const std::string & s, Context & gcontext = default_context) const;`
Description. Call `gmatch` with standard input string — base of `std::stringstream(s)` — and same context, and return result of call.

`bool gmatch_full(stream::Stream & s, Context & gcontext) const;`
Description. Start recursive descent parse with rollback, starting with node indexed as initial.
Requirements. Node indexed as initial exists.
Nodes indexed by all numbers used in grammar exist.
If data-activity flag of `context` is raised, then for each data index `i` used in grammar, size of dataset of `context` is greater or equal to `i+1`.

`bool match_full(stream::Stream & s, regexp::Context & context) const override;`
Description. If data-activity flag of `context` is raised, and main data index is not set, then immediately return `false`.
Otherwise call `gmatch_full` with same stream, and new context created from `context` as presented in general description, and return result of call.
Requirements. Node indexed as initial exists.
Nodes indexed by all numbers used in grammar exist.
If main data index is set to `i`, then size of dataset stored in grammar if greater or equal to `i+1`.
If data-activity flag of `context` is raised, then for each data index `i` used in grammar, size of dataset of `context` is greater or equal to `i+1`.

Access methods

`size_t data_size() const;`
Description. Return strored dataset size.

`bool has_main_data() const;`
Description. Return `true` $\Leftrightarrow$ main data index is set.

`size_t main_data_index() const;`
Description. Return main data index.
Requirements. Main data index is set.

`size_t main_index() const;`
Description. Return main node index.

`Node & main_node();`
Description. Return node indexed as main.
Requirements. Node indexed as main exists.

`const Node & main_node() const;`
Description. Const-analogue of previous method.

`Node & node(size_t i);`
Description. Return node indexed by `i`.
Requirements. Node indexed by `i` exists.

`const Node & node(size_t i) const;`
Description. Const-analogue of previous method.

`regexp::Regexp skip() const;`
Description. Return blank expression.

12.6 `class Node;`

Location

`grammar/node.hpp`

Description

Class of grammar nodes.
Each of 4 components of a node (an action, and three dataset functions) may be absent. An absent function is the same as the trivial function which has no effect on a dataset. An absent action is the same as the trivial action with no related data.

Default constructors and assignments, destructor

All are present, standard semantics up to default actions for tricky pointers of regular expressions.
`Node()` initializes all components as absent.

Other constructors

`Node(const Action & act);`
Description. Initializes action with `act`.
Other components are absent.

`Node(Action && act);`
Description. Move-analogue of previous constructor.

Methods for setting of components of a node

`void set_action();`
Description. Set trivial action.

`void set_action(const Action & act);`
Description. Set copy of action `act`.

`void set_action(Action && act);`
Description. Move-analogue of previous method.

`void set_post_fail();`
Description. Set trivial cleaning function.

`void set_post_fail(const dataset::Function & fun);`
Description. Set copy of `fun` as cleaning function.

`void set_post_fail(dataset::Function && fun);`
Description. Move-analogue of previous method.

`void set_post_success();`
Description. Set trivial finalization function.

`void set_post_success(const dataset::Function & fun);`
Description. Set copy of `fun` as finalization function.

`void set_post_success(dataset::Function && fun);`
Description. Move-analogue of previous method.

`void set_pre();`
Description. Set trivial preparation function.

`void set_pre(const dataset::Function & fun);`
Description. Set copy of `fun` as preparation function.

`void set_pre(dataset::Function && fun);`
Description. Move-analogue of previous method.

Methods for deletion of components of a node

`void unset();`
Description. Unset all components.

`void unset_action();`
Description. Unset action.

`void unset_post_fail();`
Description. Unset cleaning function.

`void unset_post_success();`
Description. Unset finalizaiton function.

`void unset_pre();`
Description. Unset preparation function.

Presence-checking methods

`bool has_act() const;`
Description. Return `true` ⇔ action is set.

`bool has_post_fail() const;`
Description. Return `true` ⇔ cleaning function is set.

`bool has_post_success() const;`
Description. Return `true` ⇔ finalization function is set.

`bool has_pre() const;`
Description. Return `true` ⇔ preparation function is set.

Access methods

`Action & act();`
Description. Return current action.
Requirements. Action is set.

`const Action & act() const;`
Description. Const-analogue of previous method.

`dataset::Function & post_fail();`
Description. Return current cleaning function.
Requirements. Cleaning function is set.

`const dataset::Function & post_fail() const;`
Description. Const-analogue of previous method.

`dataset::Function & post_success();`
Description. Return current finalization function.
Requirements. Finalization function is set.

`const dataset::Function & post_success() const;`
Description. Const-analogue of previous method.

`dataset::Function & pre();`
Description. Return current preparation function.
Requirements. Preparation function is set.

`const dataset::Function & pre() const;`
Description. Const-analogue of previous method.

12.7 `class ADataSet;`

Location

`grammar/context/adataset.hpp`

Namespace of the definiton

`pm::grammar::context`

Description

Class of subcontexts used as fields of grammar contexts.
Contains an activator (a tricky pointer to boolean value — an activity flag), and a dataset.
The dataset is active, is the flag is raised, and inactive, is the flag is lowered.

Default constructors and assignments, destructor

`ADataSet()` is deleted.
The rest are present, standard semantics up to default actions for tricky pointers.

Other constructors

`ADataSet(bool act, size_t size);`
Description. Direct activator to envelope of `act`.
Initialize dataset of size `size`. Direct each pointer to data to its own new empty block.

`ADataSet(const regexp::context::Activator & act, const dataset::Set & set);`
Description. Initialize dataset of the same size as `set`.
Direct all pointers of `*this` to ownees of corresponding pointers of arguments.

`ADataSet(const regexp::context::Activator & act, dataset::Set & set);`

`ADataSet(regexp::context::Activator & act, const dataset::Set & set);`

`ADataSet(regexp::context::Activator & act, dataset::Set & set);`

Description. Move-analogues of previous constructor.

Other operators

`type_abuse::DRef & operator [](size_t i);`

Description. Return `i`-th data.

Requirements. Dataset size is greater or equal to `i+1`.

`const type_abuse::DRef & operator [](size_t i) const;`

Description. Const-analogue of previous method.

Altering methods

`void detach();`

Description. Detach all pointers.

`void detach_receive(const ADataSet & adc);`

Description. Redirect activator to clone of ownee of activator of `adc`.
Call `detach_receive` of dataset of `*this` with argument — dataset of `adc`.

Requirements. Inherited from `detach_receive` of dataset: size of dataset of `adc` is greater or equal to size of dataset of `*this`.

`void detach_unset_activate();`

Description. Redirect activator to envelope of `true`.
Redirect each data pointer to its own new empty block.

`void detach_unset_deactivate();`

Description. Redirect activator to envelope of `false`.
Redirect each data pointer to its own new empty block.

Object-preserving, data-altering methods

`void receive(const ADataSet & adc) const;`

Description. Rewrite ownee of activator of `*this` with ownee of activator of `adc`.
Call `receive` of dataset of `*this` with argument — dataset of `adc`.

Requirements. Inherited from `receive` of dataset: size of dataset of `adc` is greater or equal to size of dataset of `*this`.

`void unset_activate() const;`

Description. Raise flag.
Make ownees of all data pointers empty.

`void unset_deactivate() const;`

Description. Lower flag.
Make ownees of all data pointers empty.

Object- and data-preserving methods

`ADataSet clone() const;`

Description. Return subcontext with the same dataset size as in `*this`, such that all its pointers are directed to clones of ownees of corresponding pointers of `*this`.

`void send(const ADataSet & adc) const;`

Description. Rewrite ownee of activator of `adc` with ownee of activator of `*this`.
Call `send` of dataset of `*this` with argument — dataset of `adc`.

Requirements. Inherited from `send` of dataset: size of dataset of `adc` is greater or equal to size of dataset of `*this`.

Access methods

`type_abuse::DRef & at(size_t i);`

Description. Return `i`-th data of dataset.

Requirements. Size of dataset is greater or equal to `i+1`.

`const type_abuse::DRef & at(size_t i) const;`

Description. Const-analogue of previous method.

`bool is_active() const;`
Description. Return `true` ⇔ flag is raised.

`bool is_empty(size_t i) const;`
Description. Return `true` ⇔ ownee of pointer to `i`-th data of dataset is empty.
Requirements. Size of dataset is greater or equal to `i+1`.

`bool is_inactive() const;`
Description. Return `true` ⇔ flag is lowered.

`bool is_nonempty(size_t i) const;`
Description. Return `true` ⇔ ownee of pointer to `i`-th data of dataset is nonempty.
Requirements. Size of dataset is greater or equal to `i+1`.

`size_t size() const;`
Description. Return size of dataset.

```
template<typename Value>
Value & value(size_t i) const;
```
Description. Return `i`-th data of dataset assumed to be of type `Value`.
Requirements. Size of dataset is greater or equal to `i+1`.
Ownee of pointer to `i`-th data of dataset is nonempty, and stores value of type `Value`.

Fields

`regexp::context::Activator ac;`
Description. Activator (tricky pointer to boolean flag) of dataset.

`dataset::Set dc;`
Description. Dataset.

12.8 Interface for creation of syntactic descriptions of pure actions

Location

`grammar/syntax/action.hpp`

Namespace of the definiton

`pm::grammar::syntax`

General description

This interface allows to obtain objects of type `Action`, which are syntactic descriptions of pure actions used in syntactic descriptions of grammars. The manual says to what pure actions the obtained descriptions are translated. To say it shortly, the following notation is used: `<vector>` is a vector of indexes of nodes to which elements of `vector` are translated; `<string>` is an index of a node to which a name `string` is translated.

Interface

`Action trivial_action;`
Correspondence. Trivial action.

`Action alt(const std::vector<Action> & actions);`
Correspondence. `grammar::alt(<actions>)`.

`Action alt(std::vector<Action> && actions);`
Description. Move-analogue of previous function.

`Action name(const std::string & nm);`
Correspondence. `grammar::next(<nm>)`.

`Action name(std::string && nm);`
Description. Move-analogue of previous function.

`Action re(const regexp::Regexp & r);`
Correspondence. `grammar::re(r)`.

`Action re(regexp::Regexp && r);`
Description. Move-analogue of previous function.

`Action rule(const std::vector<Action> & actions);`
Correspondence. `grammar::rule(<actions>)`.

`Action rule(std::vector<Action> && actions);`
Description. Move-analogue of previous function.

12.9 class syntax::Action;

Location

grammar/syntax/action.hpp

Namespace of the definiton

pm::grammar::syntax

Description

Class of syntactic descriptions of pure actions.
Users are not recommended to use internal interface of this class: for most cases, use a much nicer interface for creation of description of pure actions. But if anyone needs methods of this class for any reason, here they are.

Default constructors and assignments, destructor

All are present, standard semantics up to default actions for tricky pointers of regular expressions.
`Action()` initializes description of trivial action.

Types

`enum T TRIVIAL, ALTERNATIVE, NAME, REGEXP, RULE;`
Description. `TRIVIAL` means description of trivial action.
`ALTERNATIVE` means description of alternative.
`NAME` means description of forwarding.
`REGEXP` means description of terminal.
`RULE` means description of rule.

Altering methods

`void set_alt(const std::vector<Action> & actions);`
Description. Reset described action to `alt(actions)`.

`void set_alt(std::vector<Action> && actions);`
Description. Move-analogue of previous method.

`void set_name(const std::string & nm);`
Description. Reset described action to `name(nm)`.

`void set_name(std::string && nm);`
Description. Move-analogue of previous method.

`void set_re(const regexp::Regexp & r);`
Description. Reset described action to `re(r)`.

`void set_re(regexp::Regexp && r);`
Description. Move-analogue of previous method.

`void set_rule(const std::vector<Action> & actions);`
Description. Reset described action to `rule(actions)`.

`void set_rule(std::vector<Action> && actions);`
Description. Move-analogue of previous method.

`void unset();`
Description. Reset described action to trivial.

Type access methods

`T type() const;`
Description. Return sort of described action.

Presence-checking methods

`bool has_actions() const;`
Description. Return `true` ⇔ described action contains vector of node descriptions.

`bool has_name() const;`
Description. Return `true` ⇔ described action contains name.

`bool has_re() const;`
Description. Return `true` ⇔ described action contains regular expression.

Access methods

`std::vector<Action> & actions();`
Description. Return contained vector of node descriptions.
Requirements. Described action contains vector of node descriptions.

`const std::vector<Action> & actions() const;`
Description. Const-analogue of previous method.

`std::string & name();`
Description. Return contained name.
Requirements. Described action contains name.

`const std::string & name() const;`
Description. Const-analogue of previous method.

`regexp::Regexp & re();`
Description. Return contained reguar expression.
Requirements. Described action contains regular expression.

`const regexp::Regexp & re() const;`
Description. Const-analogue of previous method.

12.10 `class syntax::Grammar;`

Location

`grammar/syntax/grammar.hpp`

Namespace of the definiton

`pm::grammar::syntax`

Description

Class of syntactic grammar descriptions.
Objects of this class can be automatically translated to pure grammars.
This class contains full syntactic description of all components of a grammar: a collection of descriptions of nodes (some of which have assigned names), an initial node name, a main data name, and a blank expression.
Methods for addition of node descriptions assign a name to a description to be added. When a name is used (node addition, or inside an added node description), the name is immediately assigned to a to a full node description. This description contains subdescriptions for all node components: a pure action, a related data index (in description — a related data name), a preparation function, a finalization function, and a cleaning function. Each subdescription is assumed to be unset, if it was not set explicitly. Unset pure action is translated into a trivial action. Unset related data name is translated into no related data index. Unset dataset function is translated into a trivial function.
When a node component (the whole node) is added, if a node description with a used name exists, then the corresponding component (the whole node) is reset with new value.
When grammar description is filled, names of existing nodes may be freely used, as well as for nonexisting nodes which are intended to be filled later. According to the description above, when a name for a nonexisting node is used, an empty node description is implicitly connected to this name.

Default constructors and assignments, destructor

All are present, standard semantics up to default actions for tricky pointers of regular expressions.
`Grammar()` initialized grammar with nothing set.

Methods for action addition

`void set_action(const std::string & name, const Action & action);`
Description. Add description of pure action `action`. Assign name `name` to added node.

`void set_action(const std::string & name, Action && action);`

`void set_action(std::string && name, const Action & action);`

`void set_action(std::string && name, Action && action);`
Description. Move-analogues of previous method.

`void set_alt(const std::string & name, const std::vector<Action> & actions);`
Description. Add description of pure action `alt(actions)`. Assign name `name` to added node.

`void set_alt(const std::string & name, std::vector<Action> && actions);`

`void set_alt(std::string && name, const std::vector<Action> & actions);`

`void set_alt(std::string && name, std::vector<Action> && actions);`
Description. Move-analogues of previous method.

`void set_next(const std::string & name, const std::string & next_name);`
Description. Add description of pure action `next(next_name)`. Assign name `name` to added node.

`void set_next(const std::string & name, std::string && next_name);`

`void set_next(std::string && name, const std::string & next_name);`

`void set_next(std::string && name, std::string && next_name);`
Description. Move-analogues of previous method.

`void set_nonterminal(const std::string & name, const std::vector<std::vector<Action>> & action_matrix);`
Description. Add description of pure action `alt({rule(action_matrix[0]), rule(action_matrix[1]), ..., rule(action_matrix[k-1])})`, where `k` is the size of the external vector of `action_matrix`. Assign name `name` to added node.

`void set_nonterminal(const std::string & name, std::vector<std::vector<Action>> && action_matrix);`
`void set_nonterminal(std::string && name, const std::vector<std::vector<Action>> & action_matrix);`
`void set_nonterminal(std::string && name, std::vector<std::vector<Action>> && action_matrix);`
Description. Move-analogues of previous method.

`void set_re(const std::string & name, const regexp::Regexp & r);`
Description. Add description of pure action `re(r)`. Assign name `name` to added node.

`void set_re(const std::string & name, regexp::Regexp && r);`

`void set_re(std::string && name, const regexp::Regexp & r);`

`void set_re(std::string && name, regexp::Regexp && r);`
Description. Move-analogues of previous method.

`void set_rule(const std::string & name, const std::vector<Action> & actions);`
Description. Add description of pure action `rule(actions)`. Assign name `name` to added node.

`void set_rule(const std::string & name, std::vector<Action> && actions);`

`void set_rule(std::string && name, const std::vector<Action> & actions);`

`void set_rule(std::string && name, std::vector<Action> && actions);`
Description. Move-analogues of previous method.

Methods for other component addition

`void set_data(const std::string & name, const std::string & data);`
Description. Add related data name `name` to node description with name `name`.

`void set_data(const std::string & name, std::string && data);`

`void set_data(std::string && name, const std::string & data);`

`void set_data(std::string && name, std::string && data);`
Description. Move-analogues of previous method.

`void set_post_fail(const std::string & name, const dataset::syntax::Function & fun);`
Description. Add cleaning function `fun` to node description with name `name`.

`void set_post_fail(const std::string & name, dataset::syntax::Function && fun);`

`void set_post_fail(std::string && name, const dataset::syntax::Function & fun);`

`void set_post_fail(std::string && name, dataset::syntax::Function && fun);`
Description. Move-analogues of previous method.

`void set_post_success(const std::string & name, const dataset::syntax::Function & fun);`
Description. Add finalization function `fun` to node description with name `name`.

`void set_post_success(const std::string & name, dataset::syntax::Function && fun);`

`void set_post_success(std::string && name, const dataset::syntax::Function & fun);`

`void set_post_success(std::string && name, dataset::syntax::Function && fun);`
Description. Move-analogues of previous method.

`void set_pre(const std::string & name, const dataset::syntax::Function & fun);`
Description. Add preparation function `fun` to node description with name `name`.

`void set_pre(const std::string & name, dataset::syntax::Function && fun);`

`void set_pre(std::string && name, const dataset::syntax::Function & fun);`

`void set_pre(std::string && name, dataset::syntax::Function && fun);`
Description. Move-analogues of previous method.

Methods for setting of global components

`void set_main_data(const std::string & name);`
Description. Set main data name to `name`.

`void set_main_data(std::string && name);`
Description. Move-analogue of previous method.

`void set_main_name(const std::string & name);`
Description. Set initial node name to `name`.
If no node description is assigned to `name` assign empty description to `name`.

`void set_main_name(std::string && name);`
Description. Move-analogue of previous method.

`void set_skip(const regexp::Regexp & re);`
Description. Set blank expression to `re`.

`void set_skip(regexp::Regexp && re);`
Description. Move-analogue of previous method.

Methods for deletion of components

`void unset(const std::string & name);`
Description. Delete node name `name` and assigned node description, if they exist.

`void unset_action(const std::string & name);`
Description. If node name `name` do not exist, do nothing.
Otherwise delete description of pure action from node description with name `name`, if it was set.

`void unset_data(const std::string & name);`
Description. If node name `name` do not exist, do nothing.
Otherwise delete description of related data from node description with name `name`, if it was set.

`void unset_main_data();`
Description. Delete main data name, if it was set.

`void unset_main_name();`
Description. Delete initial node name, if it was set.
Name itself and assigned node remain existent.

`void unset_post_fail(const std::string & name);`
Description. If node name `name` do not exist, do nothing.
Otherwise delete description of cleaning function from node description with name `name`, if it was set.

`void unset_post_success(const std::string & name);`
Description. If node name `name` do not exist, do nothing.
Otherwise delete description of finalization function from node description with name `name`, if it was set.

`void unset_pre(const std::string & name);`
Description. If node name `name` do not exist, do nothing.
Otherwise delete description of preparation function from node description with name `name`, if it was set.

`void unset_skip();`
Description. Delete blank expression.

Presence-checking methods

`bool has_main_data() const;`
Description. Return `true` $\Leftrightarrow$ main data name is set.

`bool has_main_node() const;`
Description. Return `true` $\Leftrightarrow$ initial node name is set.

`bool has_node(const std::string & name) const;`
Description. Return `true` $\Leftrightarrow$ name `name` exists in grammar description.

`bool has_skip() const;`
Description. Return `true` $\Leftrightarrow$ blank expression is set.

Access methods

`std::string & main_data();`
Description. Return main data name.
Requirements. Main data name is set.

`const std::string & main_data() const;`
Description. Const-analogue of previous method.

`const std::string & main_name() const;`
Description. Return initial node name.
Requirements. Initial node name is set.

`Node & main_node();`
Description. Return initial node description.
Requirements. Initial node name is set.

`const Node & main_node() const;`
Description. Const-analogue of previous method.

`Node & node(const std::string & name);`
Description. Return description assigned to name `name`.
Requirements. Node name `name` exists in grammar description.

`const Node & node(const std::string & name) const;`
Description. Const-analogue of previous method.

`const utils::Numeration<std::string> & node_names() const;`
Description. Return numeration whose domain is the set of all node names of grammar description.

`regexp::Regexp & skip();`
Description. Return blank expression.
Requirements. Blank expression is set.

`const regexp::Regexp & skip() const;`
Description. Const-analogue of previous method.

12.11 `class syntax::Interpreter;`

Location

`grammar/syntax/interpreter.hpp`

Namespace of the definiton

`pm::grammar::syntax`

Description

Class for grammar translators.
The translation process is presented in general description, and in the description of class `Grammar`.
To perform the translation, create an object of this class, and call `interpret` with argument — syntactic grammar description. For multiple translations call `interpret` multiple times with desired descriptions.
The second method — `reset` — is currently redundant, as it is a part of the method `interpret`.

Default constructors and assignments, destructor

All are present, standard semantics.

Methods

`grammar::Grammar interpret(const Grammar & sgr);`
Description. Returns pure grammar obtained from `sgr` with translation mechanism.

`void reset();`
Description. Redundant method. Resets internal structures to initial states. Is a part of method `interpret`.

12.12 `struct syntax::Node;`

Location

`grammar/syntax/node.hpp`

Namespace of the definiton

`pm::grammar::syntax`

Description

Class of syntactic node descriptions.
Users are not recommended to use this class: for most cases, methods of the class `Grammar` are sufficient.
But if anyone for some reasons wants to use this class: it is a structure containing options for all parts of a node (pure action, related data, dataset functions).

Fields

`utils::Optional<Action> action`;
Description. If nonempty, contains syntactic description of pure action.
If empty, means trivial action.

`utils::Optional<std::string> data`;
Description. If nonempty, contains name of related data of dataset of grammar context.
If empty, means no related data.

`utils::Optional<dataset::syntax::Function> post_fail`;
Description. If nonempty, contains syntactic desctiption of cleaning function.
If empty, means trivial function.

`utils::Optional<dataset::syntax::Function> post_success`;
Description. If nonempty, contains syntactic desctiption of finalization function.
If empty, means trivial function.

`utils::Optional<dataset::syntax::Function> pre`;
Description. If nonempty, contains syntactic desctiption of preparation function.
If empty, means trivial function.