Библиотека `pm`, версия 1.0

(документация)

Ссылка на исходные тексты:
https://github.com/pellman/pm

Автор документации и библиотеки:
Владислав Подымов

16 августа 2017 г.

Содержание

1 Короткое введение

1.1 Немного имён и названий

Официальное короткое название данной библиотеки — "`pm`".

Библиотека `pm` на данный момент содержит 6 модулей с такими названиями:

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

Я[1] буду в документации обращаться к себе в третьем лице как "Автор" (с большой буквы).

"Символ" — это объект типа `char`.

"Строка" — это объект типа `std::string`.

"Стандартный поток ввода" — это объект типа `std::istream` (или производного от него, как это часто бывает).

1.2 Пара слов о библиотеке `pm`

Библиотека целиком и полностью написана на языке `C++`, местами с использованием особенностей стандарта 11.

(Не)официальная расшифровка названия библиотеки: `puzzling modules` (модули, сбивающие с толку). Об официальной расшифровке Автор умалчивает по причине её бессмысленности, так что можно считать (не)официальную расшифровку официальной.

Выбор расшифровки не случаен: функционал, предоставляемый Библиотекой,

- с одной стороны, очень похож на то, что уже присутствует в стандартных и общеизвестных нестандартных библиотеках,
- но с другой стороны, отличается местами разительно, а местами едва заметно, но всё же принципиально.

Два главных модуля, ради которых затевалась эта библиотека, — это `regexp` и `grammar`. В этих модулях реализован довольно удобный и мощный[2] аппарат разбора строк и потоков регулярными выражениями и грамматиками с попутной генерацией произвольных структур данных.

Модуль `dataset` сам по себе вряд ли будет кому-нибудь нужен. Он предоставляет основные структуры для генерации данных в модуле `grammar`.

В модулях `stream`, `type_abuse` и `utils` описаны классы и функции, семантика и назначение которых располагаются крайне близко к стандартным понятиям языка `C++`: стандартному потоку ввода (возвратный поток в `stream`), умным указателям (хитрые указатели в `type_abuse`) и другим понятиям (`utils`). Эти модули также используются в двух главных, однако, в отличие от `dataset`, могут быть интересны и сами по себе.

В свете зависимостей между понятиями модулей и уровня сложности восприятия этих модулей порядок их обсуждения в документации такой: `utils`, `stream`, `type_abuse`, `regexp`, `dataset`, `grammar`.

1.3 Пара слов о модуле `utils`

В этот модуль вошли несколько шаблонных функций и шаблонных классов общего назначения. На взгляд Автора, это те функции и классы, которые (или близкие к которым) могли бы гипотетически попасть в стандарт языка `C++`, но по тем или иным причинам в него не попали. Более точно, это:

- несколько шаблонных функций для работы с `stl`-контейнерами;[3]
- шаблонный класс `Optional` — аналог класса `std::optional`, включённого в стандарт 17;[4] объект этого класса (опционал) может содержать произвольное значение заданного типа, а также не содержать никаких значений;
- шаблонный класс `Numeration` — один частный случай так и не включённого пока ещё в стандарт шаблонного класса, реализующего взаимно-однозначного соответствие;[5] объектом класса `Numeration` (нумерацией) ставится взаимно-однозначное соответствие произвольной совокупности сравнимых[6] элементов множеству $\{0, 1, \ldots, k\}$.

1.4 Пара слов о модуле `stream`

Основной класс этого модуля — класс возвратного потока — обёртывается над стандартным потоком ввода, содержит в себе строковый буфер неограниченного размера и предоставляет возможности неформатированного чтения символов и строк (из буфера и потока) и возврата символов и строк (в буфер, но не в поток).

---

[1] Кто я и как до меня достучаться — смотрите на титульной странице.

[2] На взгляд Автора.

[3] Каждая из этих функций — это одна или пара-тройка инструкций "классической" структуры, но Автору надоело писать из раза в раз эти хоть и общеизвестные, но не совсем хорошо читаемые инструкции, так что было решено обернуть их в шаблонные функции.

[4] Наконец-таки! Все этого давно хотели, и вот в 17-м году стандартизационная махина добралась-таки. Автор решил, что чем дожидаться выхода 17-го стандарта `C++` в массы, проще написать этот класс самому.

[5] Отсутствие его в стандарте довольно странно: вопрос "почему этого нет в стандарте?" часто задавался в Интернете с давних пор, и вроде бы этот класс есть в `boost`. Но Автор хочет сохранить библиотеку максимально независимой, так что написал нужную специализацию взаимно-однозначного соответствия сам.

[6] Как в `std::map`

Автор отмечает, что хотя в стандартном потоке ввода есть методы `unget` и `putback`, основное назначение которых — возвращать символы обратно в поток, но этими методами не предоставляется никаких гарантий успешного возврата: в зависимости от того, откуда взялся используемый поток, эти методы могут успешно возвращать в поток сколько угодно чего угодно, или сколько угодно символов, ранее прочитанных из потока, или один символ, прочитанный из потока, или вообще ничего никогда. Основной класс этого модуля писался затем, чтобы можно было возвращать сколько угодно чего угодно, затем читать это заново из буфера и продолжать читать из потока.

1.5 Пара слов о модуле `type_abuse`

В этом модуле Автор всячески извращается над семантикой умных указателей, вводя понятие хитрых указателей. Классы, описанные в этом модуле, могут расцениваться как аналоги умных указателей, но работающие с несколько другими базовыми концепциями. Всё это делалось затем, чтобы

- избежать концептуальной шероховатости умных указателей, говорящей, что копированием непустого указателя разделяется владение объектом, а копирование пустого указателя — это на самом деле никакое не копирование, просто появление независимого пустого указателя; для этого в базовых концепциях модуля учитывается, что пустота бывает как одинаковой, так и разной;
- предоставить интерфейс работы с динамически создаваемыми объектами, трактуемыми как данные произвольного типа;[7] этот интерфейс активно применяется в других модулях, так что лучше отметить прямо здесь: "хитрый указатель на произвольные данные" — это объект типа `pm::type_abuse::DRef`; с учётом строгой типизации языка `C++` это, само собой, данные конкретного типа в каждый конкретный момент, но данные и их тип могут легко и просто изменяться, и всё это в рамках единого нешаблонного[8] класса.

1.6 Пара слов о модулях `regexp` и `grammar`

Популярные средства разбора регулярных выражений и грамматик и генерации чего бы то ни было из них в рамках языка `C++` (стандартный `regex`, `boost`, `lex`/`flex`/`bison` и т.д. и т.п.) — всё это коротко можно охарактеризовать словосочетанием "тихий ужас". Вроде бы эти библиотеки есть и умеют работать, и вроде как в целом все ими довольны, но в каждой из них есть неустранимые недостатки. Где-то почти нет выразительных возможностей. Где-то нет генерации данных. Где-то регулярные выражения в сочетании с генерацией выглядят абсолютно нечитаемо. Где-то позволяется разбор и генерация только внешних текстовых файлов. Где-то при встраивании в код `C++` генерируются допотопные, абослютно неинтегрируемые, неподдерживаемые и нечитаемые куски кода. Список недостатков можно продолжать долго, но итог простой — Автор написал свой код для разбора регулярных выражений и грамматик,[9] более удобный, чем существующие, для тех задач разбора строк, с которыми столкнулся. Итог — в данных модулях реализован аппарат разбора строки, стандартного потока ввода и возвратного потока (модуль `stream`) регулярными выражениями и грамматиками с попутной генерацией чего бы то ни было (более точно — данных произвольного типа из модуля `type_abuse`).

2 Структура исходных файлов

В корневой папке архива с исходными файлами находятся:

1. Подпапка `src`, содержащая исходный код библиотеки и файлы, требуемые утилитой `cmake` для автоматической сборки библиотеки.
2. Файл `README`, отсылающий читателя к данной документации и её переводу на английский язык.
3. Файл `LICENSE` — текст лицензии `GNU GPLv3`, согласно которой распространяется библиотека.
4. Файл `man_ru.pdf` — данная документация.
5. Файл `man_en.pdf` — перевод документации на английский язык.

В подпапке `src` располагаются:

1. Подподпапки `dataset`, `grammar`, `regexp`, `stream`, `type_abuse` и `utils`, в которых располагаются исходные файлы соответствующих модулей библиотеки и вспомогательные файлы `CMakeLists.txt`, требуемые утилитой `cmake` для автоматической сборки.
2. Файл `CMakeLists.txt` — основной файл для автоматической сборки библиотеки утилитой `cmake`.
3. Пустой файл `dummy.cpp`, требуемый утилитой `cmake` для своих технических нужд.

3 Лицензия

Библиотека распространяется на условиях лицензии `GNU GPLv3`. Текст лицензии можно найти, например, в файле `LICENSE` в корневой папке исходных файлов Библиотеки.

[7] Гуру `C++` спорили, спорят и будут спорить о том, хорошая или плохая особенность языка — строгая статическая типизация, и о том, можно или нельзя позволять использовать такие неустойчивые к ошибкам понятия как "данные непонятно какого типа", а Автор просто делает что хочет из того, что имеется.

[8] Это важно, потому что шаблоны - зло!

[9] С блэкджеком и китобоями.

4 Требования

Библиотека pm написана максимально независимо от чего бы то ни было. Для сборки достаточно иметь компилятор C++, настроенный на стандарт хотя бы 11. При этом полная поддержка стандарта не обязательна. Например, компилятор gcc 4.8.5 не поддерживает стандарт 11 полностью, но под ним всё собирается как надо. Сборка более ранними версиями gcc и другими компиляторами, неполностью поддерживающими стандарт 11, не гарантирована, но попробовать не запрещается: если скомпилируется, значит, работает.

Список требуемых особенностей синтаксиса языка в терминах команд утилиты cmake можно посмотреть в файле CMakeLists.txt в корневой папке исходных файлов.

Дальнейшую часть раздела требований можно пропустить, если библиотека собирается полностью и в окружении, содержащем обычный набор стандартных заголовочных файлов.

Модуль ... используется в модулях ...

Модуль dataset используется в модуле grammar.
Модуль regexp используется в модуле grammar.
Модуль stream используется в модулях grammar, regexp.
Модуль type_abuse используется в модулях dataset, grammar, regexp.
Модуль utils используется в модулях dataset, grammar, regexp.

В модуле ... используются модули ...

В модуле dataset используются модули type_abuse, utils.
В модуле grammar используются модули dataset, regexp, stream, type_abuse, utils.
В модуле regexp используются модули stream, type_abuse, utils.

Стандартный заголовочный файл ... используется в модулях ...

<cstddef> используется в модулях dataset, grammar, regexp, stream, utils.
<functional> используется в модуле type_abuse.
<initializer_list> используется в модуле dataset.
<istream> используется в модуле stream.
<list> используется в модулях dataset, regexp.
<map> используется в модулях grammar, utils.
<memory> используется в модуле type_abuse.
<set> используется в модуле regexp.
<sstream> используется в модуле regexp.
<stack> используется в модуле stream.
<string> используется в модулях dataset, grammar, regexp, stream.
<utility> используется в модулях dataset, grammar, regexp, type_abuse, utils.
<vector> используется в модулях dataset, grammar, utils.

В модуле ... используются стандартные заголовочные файлы ...

В модуле dataset используются стандартные заголовочные файлы <cstddef>, <initializer_list>, <list>, <string>, <utility>, <vector>.

В модуле grammar используются стандартные заголовочные файлы <cstddef>, <map>, <string>, <vector>.

В модуле regexp используются стандартные заголовочные файлы <cstddef>, <list>, <set>, <sstream>, <string>, <utility>.

В модуле stream используются стандартные заголовочные файлы <cstddef>, <istream>, <stack>, <string>.

В модуле type_abuse используются стандартные заголовочные файлы <functional>, <memory>, <utility>.

В модуле utils используются стандартные заголовочные файлы <cstddef>, <map>, <utility>, <vector>.

5 Инструкция по сборке

Коротко о сборке:

1. Можно просто скомилировать библиотеку из всех исходных файлов C++без дополнительных ухищрений, как это обычно делается используемым компилятором.
2. Можно легко собрать библиотеку pm или её отдельный модуль (dataset, grammar, regexp, stream) со всеми зависимостями с помощью утилиты cmake, обычным для cmake образом собрав цель с названием библиотеки или модуля.
3. Модули type_abuse и utils собирать не нужно, они состоят только из заголовочных файлов.

5.1 Сборка библиотеки без cmake

Следуйте инструкции по сборке библиотек используемого компилятора, удовлетворяющего всем немногочисленным требованиям, перечисленным в соответствующем разделе. Не забудьте включить поддержку стандарта 11. Подайте в компилятор все исходные файлы, имеющие расширение `.cpp` или `.hpp`.

5.2 Сборка библиотеки с cmake

Сборка проверена для `cmake 3.5.2` и `gcc 4.5.8`. Для других версий сборка не производилась. Отталкиваясь от документации к утилите `cmake`, Автор полагает, что самая ранняя допустимая версия `cmake`, для которой всё будет работать, — `3.1`.

С помощью `cmake` собирается статический вариант библиотеки, а также статические варианты (под)библиотек, содержащих выбранный модуль и все его зависимости. Действуйте согласно инструкции `cmake` для сборки цели `pm` (вся библиотека) или одной из целей `dataset`, `grammar`, `regexp`, `stream` (соответствующий модуль со всеми зависимостями).

Пошаговая инструкция для терминала `Linux`

Далее приведена последовательность команд, которые нужно набрать в терминале `Linux`, чтобы собрать статический вариант библиотеки в папке `folder/build` из архива `pm.1.0.tar.gz`, лежащего в папке `folder`. Чтобы собрать библиотеку из другой папки или в другой папке, замените названия папок на подходящие. Чтобы собрать библиотеку из исходных файлов (подпапки `src` распакованного архива), пропустите шаг распаковки архива и замените папку `../src` на корневую папку исходных файлов. Чтобы собрать отдельно взятый модуль со всеми зависимостями, замените на последнем шаге слово `pm` на название модуля.

В результате сборки в папке `folder/build` появится файл статической библиотеки, названный согласно конвенциям, используемым операционной системой. Например, в `Linux` статическая библиотека, собирающаяся по цели `<trg>`, имеет название `lib<trg>.a`.

Собственно последовательность команд:

1. Перейти в папку с архивом.
```
cd folder
```
2. Распаковать архив в текущую подпапку.
```
tar -xf pm.1.0.tar.gz
```
3. Создать подпапку `build`, если она ещё не создана, и перейти в неё.
```
mkdir build
cd build
```
4. Сгенерировать файлы автоматической сборки.
```
cmake ../src
```
5. Собрать библиотеку.
```
make pm
```

6 Пространства имён

Собранная библиотека — это просто все собранные модули в одном файле. Всё, что описано в библиотеке, располагается в пространстве имён `pm`.

Все определения модуля `M` располагаются в пространстве имён `pm::M`. Например, доступ к основному классу `Regexp` модуля `regexp` выглядит так:

`pm::regexp::Regexp`.

Внутри пространства имён модуля могут иметься вложенные пространства имён. Например, некоторые определения модуля `regexp`, определяющие контекст разбора возвратного потока, располагаются в пространстве имён

`pm::regexp::context`.

Подробное описание пользовательского интерфейса модулей приведено в соответствующих разделах описания модулей.

В документации соблюдаются следующие условности:

1. Описываемый интерфейс располагается в пространстве имён модуля, если не сказано иного.
2. При описании интерфейса:
   - опускается пространство имён всех компонентов описания, располагающихся в том же пространстве, что и описываемый интерфейс;
   - частично указывается пространство имён всех компонентов описания, располагающихся в подпространствах пространства имён интерфейса: если интерфейс располагается в пространстве имён `a::b`, а описываемый компонент (скажем, тип `T`) — в пространстве имён `a::b::c::d`, то отсылка к этому компоненту выглядит как `c::d::T`;

- почти польностью указывается пространство имён компонентов описания, располагающихся в других модулях библиотеки: опускается начало "`pm::`", остальная часть пространства имён указывается полностью;
- полностью указывается пространство имён всех остальных компонентов описания.

7 Интерфейс модуля `utils`

Все определения этого модуля располагаются в пространстве имён `pm::utils`.

7.1 `template<typename Element> class Numeration;`

Расположение

`utils/numeration.hpp`

Описание

Нумерация — это объект класса `Numeration<Element>`.
В каждый момент жизни нумерацией определено биективное отображение совокупности элементов типа `Element` в множество $\{0, 1, \ldots, k-1\}$, где k — количество отображаемых элементов.
Прообраз — это отображаемый элемент.
Образ — это число, в которое отображается прообраз.
Область определения — это совокупность всех прообразов нумерации.
При добавлении элементов в область определения все образы элементов, существовавших в области, остаются неизменными.
При удалении элементов из области определения вид получающегося отображения не специфицирован.

Требования к аргументам шаблона

Объекты типа `Element` должны быть сравнимы оператором `<`.
В зависимости от применяемых методов могут потребоваться копирование и сдвиг объектов типа `Element`.

Конструкторы и присваивания по умолчанию, деструктор

Все присутствуют, стандартная семантика.
`Numeration()` инициализирует нумерацию пустой областью определения.

Методы, изменяющие состояние объекта

`size_t add(const Element & el);`
Описание. Добавить `el` в область определения, если не содержится в ней.
В любом случае вернуть образ `el`.

`size_t add(Element && el);`
Описание. Move-аналог предыдущего метода.

`void add(const Numeration<Element> & num);`
Описание. Добавить в область определения все прообразы нумерации `num`

`void add(Numeration<Element> && num);`
Описание. Move-аналог предыдущего метода.

`void clear();`
Описание. Очистить область определения.

`void remove(const Element & el);`
Описание. Удалить `el` из области определения.

Методы доступа

`const Element & element_at(size_t i) const;`
Описание. Вернуть прообраз числа `i`.
Требования. Размер области определения больше либо равен `(i+1)`.

`bool has(const Element & el) const;`
Описание. Вернуть `true` $\Leftrightarrow$ `el` содержится в области определения.

`size_t index_of(const Element & el) const;`
Описание. Вернуть образ элемента `el`

Требования. Элемент `el` содержится в области определения.

`size_t size() const;`
Описание. Вернуть размер области определения.

`std::vector<Element>::const_iterator begin() const;`
Описание. Вернуть итератор на первый элемент области определения (для "`range-based for`").

`std::vector<Element>::const_iterator end() const;`
Описание. Вернуть итератор на конец области определения (для "`range-based for`").

7.2 `template<typename Value> class Optional;`

Расположение

`utils/optional.hpp`

Описание

Опционал — это объект класса `Optional<Value>`.
В каждый момент жизни опционал

- либо непуст, и содержит одно значение типа `Value`,
- либо пуст, и не содержит никаких значений.

Требования к аргументам шаблона

В зависимости от применяемых методов могут потребоваться копирование и сдвиг объектов типа `Value`.

Конструкторы и присваивания по умолчанию, деструктор

Все присутствуют, стандартная семантика.
`Optional()` инициализирует пустой опционал.

Другие конструкторы и присваивания

`Optional(const Value & value);`
Описание. Инициализировать непустой опционал, хранящий копию значения `value`.

`Optional(Value && value);`
Описание. Move-аналог предыдущего конструктора.

`Optional & operator =(const Value & value);`
Описание. Записать в опционал копию значения `value`.

`Optional & operator =(Value && value);`
Описание. Move-аналог предыдущего присваивания

Методы, изменяющие состояние объекта

`void unset();`
Описание. Опустошить опционал.

Методы доступа

`bool has_value() const;`
Описание. Вернуть `true` $\Leftrightarrow$ опционал непуст.

`Value & value();`
Описание. Вернуть хранимое значение.
Требования. Опционал непуст.

`const Value & value() const;`
Описание. Const-аналог предыдущего метода.

7.3 Функции работы с `stl`-контейнерами

Расположение

`utils/container_functions.hpp`

Общие условия

Во всех прототипах шаблонных функций далее
- `Container` — тип `stl`-контейнера или аналогичный ему по интерфейсу;
- `Element` — тип элементов контейнера.

Собственно функции

```
template<typename Container>
void append(Container & where, const Container & who);
```
Описание. Добавить в конец контейнера `where` элементы контейнера `who`.
Требования. Выражение `where.append(where.end(), who.begin(), who.end())` допустимо.

```
template<typename Container>
bool disjoint(const Container & X, const Container & Y);
```
Описание. Вернуть `true` ⇔ контейнеры `X` и `Y` не содержат элементов `x`, `y`, таких что `x == y`.
Требования. Тип `Container`
- допускает перебор элементов с помощью "`range-based for`" и
- содержит метод `find(const Element &)`.

```
template<typename Container>
void erase_index(Container & container, size_t ind);
```
Описание. Удалить из контейнера `ind`-й элемент, то есть элемент по смещению `container.begin() + ind`.
Требования. Выражение `container.begin() + ind` допустимо.

```
template<typename Container>
bool intersects(const Container & X, const Container & Y);
```
Описание. Вернуть `true` ⇔ в контейнерах `X` и `Y` содержатся элементы `x`, `y`, такие что `x == y`.
Требования. Тип `Container`
- допускает перебор элементов с помощью "`range-based for`" и
- содержит метод `find(const Element &)`.

```
template<typename Element, typename Container>
bool is_in(const Element & element, const Container & container);
```
Описание. Вернуть `true` ⇔ контейнер содержит элемент `x`, такой что `x == element`.
Требования. Выражение `container.find(element)` допустимо.
Элементы сравнимы оператором `==`.

```
template<typename Element, typename Container>
bool isnt_in(const Element & element, const Container & container);
```
Описание. Вернуть `true` ⇔ контейнер не содержит элементов `x`, таких что `x == element`.
Требования. Выражение `container.find(element)` допустимо.
Элементы сравнимы оператором `==`.

```
template<typename Container>
void merge(Container & where, const Container & who);
```
Описание. Вставить элементы `who` в совокупность элементов `where`.
Требования. Контейнер содержит методы `begin()`, `end()`, а также бинарный метод `insert`, принимающий итераторы на начало и конец спектра вставляемых значений.

8 Интерфейс модуля `stream`

Все определения этого модуля располагаются в пространстве имён `pm::stream`.

8.1 `class Stream;`

Расположение

`stream/stream.hpp`

Описание

Возвратный поток — это объект класса `Stream`.

Возвратный поток содержит в себе стандартный поток ввода (связанный поток) и строковый буфер неограниченного объёма и предназначен для чтения и возврата символов и строк, аналогичного неформатированному чтению символов и последовательностей символов из стандартного потока ввода и возврату их в стандартный поток ввода в предположении о том, что методы возврата (`unget`, `putback`) всегда отрабатывают успешно.

Буфер разбит на блоки (стек строк) одинаковой фиксированной длины. Выбор длины блоков при желании можно делать при инициализации возвратного потока через особый конструктор. Этот выбор никак не влияет на функционал, он влияет только на производительность операций чтения и возврата.

Чтение из возвратного потока разбивается на два этапа: на первом происходит чтение из буфера, и если буфер опустошён и не прочитано нужное количество символов, то на втором производится чтение из связанного потока. Первый этап чтения всегда проходит успешно. Чтение из возвратного потока может быть неуспешным, но только по причине неуспешного чтения из связанного потока.

Символы возвращаются только в буфер и так, чтобы при последующем чтении соблюдался следующий порядок символов:

- возвращённые символы и строки — от последних к первым (по принципу стека);
- внутри одной помещённой строки — от первого символа к последнему.

Возврат символов и строк в возвратный поток всегда успешен.

Конструкторы и присваивания по умолчанию, деструктор

Все такие конструкторы и приваивания удалены.

Деструктор — обычный.

Другие конструкторы

`Stream(std::istream & stream);`

Описание. Связать `*this` с потоком `stream` через ссылку.
Завести пустой буфер с размером блока `100`.

`Stream(std::istream & stream, size_t buffer_size);`

Описание. Связать `*this` с потоком `stream` через ссылку.
Завести пустой буфер с размером блока `buffer_size`.

Методы, изменяющие состояние объекта

`bool get(char & c);`

Описание. Попытаться прочитать символ из `*this`.
Если символ успешно прочитан, то записать его в переменную `c`.
В противном случае не изменять переменную `c`.
В любом случае вернуть `true` ⇔ символ успешно прочитан (и записан в `c`).

`size_t get(std::string & s, size_t size);`

Описание. Прочитать как можно больше, но не более `size` символов из `*this`.
Записать прочитанную последовательность в строку `s` (если не прочитано ни одного символа, то опустошить строку).
Вернуть количество прочитанных символов.

`bool get_strict(std::string & s, size_t size);`

Описание. Попытаться прочитать в точности `size` символов из `*this`.
В случае успеха записать прочитанную последовательность в строку `s`.
В случае неуспеха вернуть все реально прочитанные символы в буфер и не изменять строку `s`.
В любом случае вернуть `true` ⇔ `size` символов успешно прочитано (и записано в `s`).

`bool skip();`

Описание. Попытаться прочитать ровно один символ из `*this` и никуда его не записывать.
Вернуть `true` ⇔ символ успешно прочитан.

`size_t skip(size_t size);`

Описание. Прочитать как можно больше, но не более `size` символов из `*this` и никуда их не записывать.
Вернуть количество прочитанных символов.

`bool skip_strict(size_t size);`

Описание. Попытаться прочитать ровно `size` символов из `*this` и никуда их не записывать.
В случае неуспеха вернуть реально прочитанные символы в буфер.
В любом случае вернуть `true` ⇔ нужное число символов успешно прочитано.

`void unget(char c);`

Описание. Вернуть символ `c` в буфер.

`void unget(const std::string & s);`

Описание. Вернуть строку `s` в буфер.

Методы доступа

`bool finished() const;`

Описание. Вернуть `true` ⇔ буфер пуст и в связанном потоке возникла ошибка, то есть поднят хотя бы один из флагов ошибок.[10] При поднятии любого такого флага `*this` полагает, что дальнейшее чтение из связанного потока невозможно.

9 Интерфейс модуля `type_abuse`

Все определения этого модуля располагаются в пространстве имён `pm::type_abuse`.

9.1 Общее описание

Небольшое напоминание. В языке `C++` существуют две популярные концепции указателей:

- чистый указатель на тип `T`, то есть указатель типа `T *`;
- умный указатель на тип `T` — общее название стандартных типов указателей со встроенной сборкой мусора и характерными особенностями управления множественным указыванием на один объект, например, `std::shared_ptr<T>` и `std::unique_ptr<T>`.

В данном модуле описывается альтернативная концепция указателей, называемая далее “хитрые указатели”.[11]

9.1.1 Блоки данных

Блок (данных) — это особая сущность, на которую указывает хитрый указатель. Блоки могут быть устроены совершенно по-разному, точное техническое устройство блоков не обсуждается.

Блок может быть

- непустым, и содержать (хранить) данные (объект, значение);
- пустым, и не содержать данных.

Данные блока — это данные, хранимые в блоке.

Хитрый указатель в каждый момент жизни указывает ровно на один блок данных (связан с этим блоком; владеет этим блоком).

Блок указателя — это блок, связанный с указателем.

Концептуальное отличие хитрых указателей от чистых и умных состоит в том, что не существует пустых хитрых указателей (наподобие `nullptr`), вместо них существуют хитрые указатели, указывающие на пустые блоки данных. Можно создавать много различных пустых блоков данных (в отличие от единого `nullptr` для всех указателей). На один пустой блок данных может указывать много различных хитрых указателей. В пустой блок данных можно записать данные — он становится непустым, и хранимое значение становится доступным из всех указывающих на него хитрых указателей.

Обёртка над объектом — это новый непустой блок, хранящий этот объект.

Блок является адресным, если можно получить чистый указатель с адресом хранящихся в нём данных (если блок непуст).[12] Этот адрес называется основой блока. Получаемый чистый указатель имеет заданный тип (скажем, `T *`), и в этом случае тип `T` называется базовым.

Обёртка над основой — это новый непустой адресный блок с этой основой.

Адресные блоки хорошо подходят для управления динамически создаваемыми объектами базового и производных типов — как и в чистых и умных указателях. Неадресные блоки не рекомендуются для использования в таких целях, их основное назначение — оперирование данными произвольных явно указываемых типов.

9.1.2 Сборка мусора

В хитрые указатели, как и в умные, встроена сборка мусора:[13]

- если непустым блоком не владеет ни один хитрый указатель, то вызывается деструктор данных, хранимых в этом блоке;
- если при вызове метода указателя текущие данные непустого блока “уходят в никуда”, то они уничтожаются вызовом деструктора.

[10] Для тех кто в танке, их ровно три: попытка чтения за пределами конца файла; логическая ошибка ввода/вывода; ошибка чтения/записи при вводе/выводе. Поднятие этих флагов, как правило, означает, что поток сломался или закончился.

[11] По аналогии с умными указателями. Названа так в силу отсутствия фантазии.

[12] По аналогии с методом `get` умных указателей.

[13] Более точно, в них используется сборка мусора, предоставляемая типами `std::shared_ptr` и иногда `std::unique_ptr`.

В частности, это означает, что не следует явно или неявно уничтожать данные блока в обход средств, предоставляемых хитрыми указателями (например, явно вызывать деструктор, освобождать память, использовать одну основу в нескольких блоках, использовать основу адресного блока в умных указателях и т.д.).

Данные непустого адресного блока уничтожаются вызовом деструктора разыменования основы этого блока.

9.1.3 Владение блоком

Владение блоком разделяется между хитрыми указателями одинакового типа через копирование, сдвиг, присваивание и специальные методы хитрых указателей согласно документации, и никак иначе.[14] Попытки передать владение блоком другими (недокументированными) способами приводят к неспецифицированному поведению — как правило, либо к непредсказуемому дублированию данных с некорректным разделением владения, либо к ошибке "segmentation fault" в связи с обращением к освобождённой памяти.

9.1.4 Операции над блоками

Запись данных `v` в блок приводит к тому, что блок начинает хранить значение `v`.

Вставка основы (значения чистого указателя) в адресный блок приводит к тому, что основой блока становится в точности значение вставляемого указателя, и блок начинает хранить данные, лежащие по адресу этого указателя.

Перезапись блока `A` блоком `B` выглядит так:

- если блок `B` пуст, то блок `A` опустошается;
- если блок `B` непуст, то в блок `A` записывается копия данных, хранимых блоком `B`.

Возможен один из трёх способов записи копии:

1. Способ по умолчанию: выделение памяти и применение конструктора копирования или сдвига;
2. Явно обозначенный способ: вставка основы, получаемой с помощью специального метода базового типа;
3. Явно обозначенный способ: применение оператора присваивания.

Клонирование блока приводит к тому же результату, что и создание нового пустого блока с немедленной перезаписью новосозданного блока клонируемым блоком. Клон блока — это новый блок, создаваемый при клонировании исходного блока.

9.1.5 Операции над хитрыми указателями

Перенаправление хитрого указателя с одного блока на другой — это устранение его связи с одним блоком (без изменения этого блока, за исключением уничтожения при сборке мусора) и установление (разделение) связи с другим блоком.

Открепление хитрого указателя — это перенаправление его на клона текущего связанного с указателем блока.

Клон хитрого указателя — это хитрый указатель того же типа, указывающий на клона блока, связанного с исходным указателем.

9.1.6 Разнообразие хитрых указателей

Различия между типами хитрых указателей состоят в том,

1. является ли связанный блок адресным;
2. есть ли ограничения на базовый тип адресного блока;
3. можно ли связать хитрый указатель с пустым блоком;
4. можно ли наполнить/опустошить блок или сменить тип данных блока после инициализации блока.

Выбор конкретного типа хитрого указателя пользователем зависит, например, от следующих параметров:

- какие возможности указателей будут использоваться (например, если в программе будут требоваться чистые указатели на хранимые объекты и аппарат доступа к объектам производного класса через чистый указатель на базовый, то следует использовать хитрые указатели, связываемые с адресными блоками);
- насколько важна производительность (чем шире возможности хитрого указателя, тем медленнее и затратнее по памяти он будет работать).

9.2 `template<typename Base> class BRef;`

Расположение

`type_abuse/bref.hpp`

[14] Примерно как в `std::shared_ptr`.

Описание

Хитрый указатель этого типа связывается только с непустыми адресными блоками с базовым типом `Base`. Тип хранимых данных всегда неизменен, и это тип `Base`.

Исключение: можно хранить данные типов, производных от `Base`, если соблюдаются следующие условия:

1. Запись данных в блок, связанный с указателем, осуществляется только вставкой основы.
2. Вызов деструктора разыменования основы корректно уничтожает хранимый объект.
3. Не применяются методы хитрого указателя, создающие копии хранимого объекта.

Требования к аргументу шаблона

В зависимости от используемых методов от типа `Base` может требоваться наличие конструктора без аргументов, конструктора присваивания, конструктора сдвига, стандартного оператора присваивания, стандартного оператора присваивания со сдвигом и других конструкторов, запрашиваемых пользователем.

Конструкторы и присваивания по умолчанию, деструктор

`BRef();`
Описание. Направить `*this` на обёртку над `Base()`.

`BRef(const BRef<Base> & ref);`
Описание. Направить `*this` на блок указателя `ref`.

`BRef(Bref<Base> && ref);`
Описание. Move-аналог предыдущего конструктора.

`BRef & operator =(const BRef<Base> & ref);`
Описание. Перенаправить `*this` на блок указателя `ref`.
Вернуть `*this`.

`BRef & operator =(BRef<Base> && ref);`
Описание. Move-аналог предыдущего присваивания.

`~BRef();`
Описание. Обычный деструктор.

Другие конструкторы

`BRef(Base * ptr);`
Описание. Если `ptr == nullptr`, то направить `*this` на обёртку над `Base()`.
Иначе направить `*this` на обёртку над основой `ptr`.
Требования. Указатель `ptr` разыменовываем.
Данные по адресу `ptr` не должны явно или неявно уничтожаться никем, кроме хитрых указателей, разделяющих владение новосозданным блоком.

`BRef(const Base & v);`
Описание. Направить `*this` на обёртку над копией значения `v`.

`BRef(Base && v);`
Описание. Move-аналог предыдущего конструктора.

Другие операторы

`Base * operator ->() const;`
Описание. Аналог такого же оператора для чистых указателей.
Оператор доступа к полям и методам разыменования основы блока указателя `*this`.

`Base & operator *() const;`
Описание. Аналог такого же оператора для обычных указателей.
Вернуть разыменование основы блока указателя `*this`.

Методы, изменяющие состояние объекта

`void detach();`
Описание. Открепить указатель `*this`.

`void detach_own(Base * ptr);`
Описание. Перенаправить `*this` на обёртку над основой `ptr`.
Требования. Указатель `ptr` разыменовываем.
Данные по адресу `ptr` не должны явно или неявно уничтожаться никем, кроме хитрых указателей, разделяющих владение новосозданным блоком.

`void detach_receive(const BRef<Base> & ref);`

Описание. Перенаправить `*this` на клона блока указателя `ref`.

```
template<typename ... Args>
detach_set(Args && ... args);
```

Описание. Перенаправить `*this` на обёртку над `Base(args ...)`.

Методы, сохраняющие состояние объекта и изменяющие хранимые данные

```
void receive(const BRef<Base> & ref) const;
```

Описание. Перезаписать блок указателя `*this` блоком указателя `ref` через присваивание.

```
template<typename ... Args>
void set(Args && ... args) const;
```

Описание. Записать объект `Base(args ...)` в блок указателя `*this` через присваивание.

Методы, сохраняющие состояние объекта и хранимые данные

```
BRef<Base> clone() const;
```

Описание. Вернуть клона указателя `*this`.

```
void send(const BRef<Base> & ref) const;
```

Описание. Перезаписать блок указателя `ref` блоком указателя `*this` через присваивание.

Методы доступа

```
Base * get() const;
```

Описание. Вернуть основу блока указателя `*this`.

Требования. Данные по возвращаемому адресу не должны явно или неявно уничтожаться никем, кроме хитрых указателей, разделяющих владение блоком указателя `*this`.

```
Base * get_copy() const;
```

Описание. Создать копию данных блока указателя `*this` и вернуть чистый указатель на неё.

```
Base & val() const;
```

Описание. Вернуть разыменование основы блока указателя `*this`.

Статические методы

```
template<typename ... Args>
static BRef<Base> create(Args && ... args);
```

Описание. Вернуть хитрый указатель, связанный с обёрткой над `Base(args ...)`.

9.3 class CDRef;

Расположение

`type_abuse/cdref.hpp`

Описание

Хитрый указатель этого типа связывается с пустыми и непустыми неадресными блоками.
Пустота/непустота блока и тип данных, хранимых в непустом блоке, определяются единожды в момент инициализации блока и остаются неизменными до уничтожения блока.

Требования к используемым типам хранимых данных

В зависимости от используемых методов может требоваться наличие конструкторов копирования и сдвига для данных хранимого типа, а также других конструкторов, указываемых пользователем.

Конструкторы и присваивания по умолчанию, деструктор

```
CDRef();
```

Описание. Направить `*this` на новый пустой блок.

```
CDRef(const CDRef & ref);
```

Описание. Направить `*this` на блок указателя `ref`.

```
CDRef(CDRef && ref);
```

Описание. Move-аналог предыдущего конструктора.

```
CDRef & operator =(const CDRef & ref);
```

Описание. Перенаправить `*this` на блок указателя `ref`.
Вернуть `*this`.

```
CDRef & operator =(CDRef && ref);
```
Описание. Move-аналог предыдущего присваивания.

```
~CDRef();
```
Описание. Обычный деструктор.

Другие операторы

```
operator bool() const;
```
Описание. Вернуть `true` ⇔ блок указателя `*this` непуст.

Методы, изменяющие состояние объекта

```
void detach();
```
Описание. Открепить указатель `*this`.

```
void detach_receive(const CDRef & ref);
```
Описание. Перенаправить `*this` на клона блока указателя `ref`.

```
template<typename Value, typename ... Args>
void detach_set(Args && ... args);
```
Описание. Перенаправить `*this` на обёртку над `Value(args ...)`.

Методы, сохраняющие состояние объекта

```
CDRef clone() const;
```
Описание. Вернуть клона указателя `*this`.

Методы доступа

```
bool empty() const;
```
Описание. Вернуть `true` ⇔ блок указателя `*this` пуст.

```
bool nonempty() const;
```
Описание. Вернуть `true` ⇔ блок указателя `*this` непуст.

```
template<typename Value>
Value & val() const;
```
Описание. Вернуть ссылку на данные блока указателя `*this`, трактуемые как данные типа `Value`.
Требования. Блок указателя `*this` непуст, хранимые данные имеют тип `Value`.

Статические методы

```
template<typename Value, typename ... Args>
static CDRef create(Args && ... args);
```
Описание. Вернуть хитрый указатель, связанный с обёрткой над `Value(args ...)`.

9.4 `template<typename Base>` `class CRef;`

Расположение

`type_abuse/cref.hpp`

Описание

Указатель связывается с пустыми и непустыми адресными блоками с базовым типом `Base`.
Пустота/непустота блока и тип данных, хранимых в непустом блоке, определяются единожды в момент инициализации блока и остаются неизменными до уничтожения блока.

Требования к аргументу шаблона

В зависимости от вызываемых методов от типа `Base` может требоваться наличие конструкторов копирования и сдвига для данных хранимого типа, а также других конструкторов, явно указываемых пользователем.
Копирование данных производится вызовом клонирующего метода в типе `Base`: метода `Base * Base::clone()`, возвращающего указатель на новую копию объекта, для которого вызван этот метод.
Если требуется копирование объектов производных типов, то это, как правило, подразумевает виртуальность клонирующего метода и его перегрузку в производных классах.

Конструкторы и присваивания по умолчанию, деструктор

`CRef();`
Описание. Направить `*this` на новый пустой блок.

`CRef(const CRef<Base> & ref);`
Описание. Направить `*this` на блок указателя `ref`.

`CRef(CRef<Base> && ref);`
Описание. Move-аналог предыдущего конструктора.

`CRef & operator =(const CRef<Base> & ref);`
Описание. Перенаправить `*this` на блок указателя `ref`.
Вернуть `*this`.

`CRef & operator =(CRef<Base> && ref);`
Описание. Move-аналог предыдущего присваивания.

`~CRef();`
Описание. Обычный деструктор.

Другие конструкторы

`CRef(Base * ptr);`
Описание. Направить `*this` на обёртку над основой `ptr`.
Требования. Указатель `ptr` разыменовываем.
Данные по адресу `ptr` не должны явно или неявно уничтожаться никем, кроме хитрых указателей, разделяющих владение новосозданным блоком.

Другие операторы

`Base * operator ->() const;`
Описание. Аналог такого же оператора для обычных указателей.
Оператор доступа к полям и методам разыменования основы блока указателя `*this`.
Требования. Блок указателя `*this` непуст.

`Base & operator *() const;`
Описание. Аналог такого же оператора для обычных указателей.
Вернуть разыменование основы блока указателя `*this`.
Требования. Блок указателя `*this` непуст.

`operator bool() const;`
Описание. Вернуть `true` $\Leftrightarrow$ блок указателя `*this` непуст.

Методы, изменяющие состояние объекта

`void detach();`
Описание. Открепить указатель `*this`.

`void detach_own(Base * ptr);`
Описание. Перенаправить `*this` на обёртку над основой `ptr`.
Требования. Указатель `ptr` разыменовываем.
Данные по адресу `ptr` не должны явно или неявно уничтожаться никем, кроме хитрых указателей, разделяющих владение новосозданным блоком.

`void detach_receive(const CRef<Base> & ref);`
Описание. Перенаправить `*this` на клона блока указателя `ref`.

```
template<typename Value = Base, typename ... Args>
void detach_set(Args && ... args);
```
Описание. Перенаправить `*this` на обёртку над `Value(args ...)`.

Методы, сохраняющие состояние объекта

`CRef<Base> clone() const;`
Описание. Вернуть клона указателя `*this`.

Методы доступа

`bool empty() const;`
Описание. Вернуть `true` ⇔ блок указателя `*this` пуст.

`Base * get() const;`
Описание. Вернуть основу блока указателя `*this`.
Требования. Блок указателя `*this` непуст.
Данные по возвращаемому адресу не должны явно или неявно уничтожаться никем, кроме хитрых указателей, разделяющих владение блоком указателя `*this`.

`Base * get_copy() const;`
Описание. Создать копию данных блока указателя `*this` и вернуть чистый указатель на неё.
Требования. Блок указателя `*this` непуст.

`bool nonempty() const;`
Описание. Вернуть `true` ⇔ блок указателя `*this` непуст.

```
template<typename Value = Base>
Value & val() const;
```
Описание. Вернуть ссылку на данные блока указателя `*this`, трактуемые как данные типа `Value` посредством `static_cast`.
Требования. Блок указателя `*this` непуст.
Трактовка данных корректна.

Статические методы

```
template<typename Value = Base, typename ... Args>
static CRef<Base> create(Args && ... args);
```
Описание. Вернуть хитрый указатель, связанный с обёрткой над `Value(args ...)`.

9.5 class DRef;

Расположение

`type_abuse/dref.hpp`

Описание

Класс, объекты которого в других частях документации называются хитрыми указателями на произвольные данные.

Указатель связывается с пустыми и непустыми неадресными блоками. Пустота/непустота блока и тип данных, хранимых в непустом блоке, могут изменяться на протяжении жизни блока.

Требования к используемым типам хранимых данных

В зависимости от вызываемых методов может требоваться наличие конструкторов копирования и сдвига и других для данных хранимого типа.

Конструкторы и присваивания по умолчанию, деструктор

`DRef();`
Описание. Направить `*this` на новый пустой блок.

`DRef(const DRef & ref);`
Описание. Направить `*this` на блок указателя `ref`.

`DRef(DRef && ref);`
Описание. Move-аналог предыдущего конструктора.

`DRef & operator =(const DRef & ref);`
Описание. Перенаправить `*this` на блок указателя `ref`.
Вернуть `*this`.

`DRef & operator =(DRef && ref);`
Описание. Move-аналог предыдущего присваивания.

`~DRef();`
Описание. Обычный деструктор.

Другие операторы

`operator bool() const;`
Описание. Вернуть `true` ⇔ блок указателя `*this` непуст.

Методы, изменяющие состояние объекта

`void detach();`
Описание. Открепить указатель `*this`.

`void detach_receive(const DRef & ref);`
Описание. Перенаправить `*this` на клона блока указателя `ref`.

```
template<typename Value, typename ... Args>
void detach_set(Args && ... args);
```
Описание. Перенаправить `*this` на обёртку над `Value(args ...)`.

`void detach_unset();`
Описание. Перенаправить `*this` на новый пустой блок.

Методы, сохраняющие состояние объекта и изменяющие хранимые данные

`void receive(const DRef & ref) const;`
Описание. Перезаписать блок указателя `*this` блоком указателя `ref`.

```
template<typename Value, typename ... Args>
void set(Args && ... args) const;
```
Описание. Записать данные `Value(args ...)` в блок указателя `*this`.

`void swap(const DRef & ref) const;`
Описание. Поменять местами данные, хранимые в блоках указателей `*this` и `ref`.

`void unset() const;`
Описание. Опустошить блок указателя `*this`.

Методы, сохраняющие состояние объекта и хранимые данные

`DRef clone() const;`
Описание. Вернуть клона указателя `*this`.

`void send(const DRef & ref) const;`
Описание. Перезаписать блок указателя `ref` блоком указателя `*this`.

Методы доступа

`bool empty() const;`
Описание. Вернуть `true` ⇔ блок указателя `*this` пуст.

`bool nonempty() const;`
Описание. Вернуть `true` ⇔ блок указателя `*this` непуст.

```
template<typename Value>
Value & val() const;
```
Описание. Вернуть ссылку на данные блока указателя `*this`, трактуемые как данные типа `Value`.
Требования. Блок указателя `*this` непуст, хранимые данные имеют тип `Value`.

Статические методы

```
template<typename Value, typename ... Args>
static DRef create(Args && ... args);
```
Описание. Вернуть хитрый указатель, связанный обёрткой над `Value(args ...)`.

9.6 `template<typename Base> class Ref;`

Расположение

`type_abuse/ref.hpp`

Описание

Указатель связывается с пустыми и непустыми адресными блоками с базовым типом `Base`.

Пустота/непустота блока и тип данных, хранимых в непустом блоке, могут изменяться на протяжении жизни блока.

Требования к аргументу шаблона

В зависимости от вызываемых методов может требоваться наличие конструкторов копирования и сдвига и других для данных хранимого типа.

Копирование данных производится вызовом клонирующего метода в типе `Base`: метода `Base * Base::clone()`, возвращающего указатель на новую копию объекта, для которого вызван этот метод.

Если требуется копирование объектов производных типов, то это, как правило, подразумевает виртуальность клонирующего метода и его перегрузку в производных классах.

Конструкторы и присваивания по умолчанию, деструктор

`Ref();`
Описание. Направить `*this` на новый пустой блок.

`Ref(const Ref<Base> & ref);`
Описание. Направить `*this` на блок указателя `ref`.

`Ref(Ref<Base> && ref);`
Описание. Move-аналог предыдущего конструктора.

`Ref & operator =(const Ref<Base> & ref);`
Описание. Перенаправить `*this` на блок указателя `ref`.
Вернуть `*this`.

`Ref & operator =(Ref<Base> && ref);`
Описание. Move-аналог предыдущего присваивания.

`~Ref();`
Описание. Обычный деструктор.

Другие конструкторы

`Ref(Base * ptr);`
Описание. Направить `*this` на обёртку над основой `ptr`.
Требования. Указатель `ptr` разыменовываем.
Данные по адресу `ptr` не должны явно или неявно уничтожаться никем, кроме хитрых указателей, разделяющих владение новосозданным блоком.

Другие операторы

`Base * operator ->() const;`
Описание. Аналог такого же оператора для обычных указателей.
Оператор доступа к полям и методам разыменования основы блока указателя `*this`.
Требования. Блок указателя `*this` непуст.

`Base & operator *() const;`
Описание. Аналог такого же оператора для обычных указателей.
Вернуть разыменование основы блока указателя `*this`.
Требования. Блок указателя `*this` непуст.

`operator bool() const;`
Описание. Вернуть `true` $\Leftrightarrow$ блок указателя `*this` непуст.

Методы, изменяющие состояние объекта

`void detach();`
Описание. Открепить указатель `*this`.

`void detach_own(Base * ptr);`
Описание. Перенаправить `*this` на обёртку над основой `ptr`.
Требования. Указатель `ptr` разыменовываем.
Данные по адресу `ptr` не должны явно или неявно уничтожаться никем, кроме хитрых указателей, разделяющих владение новосозданным блоком.

`void detach_receive(const Ref<Base> & ref);`

Описание. Перенаправить `*this` на клона блока указателя `ref`.

```
template<typename Value = Base, typename ... Args>
void detach_set(Args && ... args);
```

Описание. Перенаправить `*this` на обёртку над `Value(args ...)`.

```
void detach_unset();
```

Описание. Перенаправить `*this` на новый пустой блок.

Методы, сохраняющие состояние объекта и изменяющие хранимые данные

```
void own(Base * ptr) const;
```

Описание. Вставить основу `ptr` в блок указателя `*this`.

```
void receive(const Ref<Base> & ref) const;
```

Описание. Перезаписать блок указателя `*this` блоком указателя `ref`.

```
template<typename Value = Base, typename ... Args>
void set(Args && ... args) const;
```

Описание. Записать объект `Value(args ...)` в блок указателя `*this`.

```
void swap(const Ref<Base> & ref) const;
```

Описание. Поменять местами данные, хранимые в блоках указателей `*this` и `ref`.

```
void swap(std::unique_ptr<Base> & ptr) const;
```

Описание. Поменять местами данные, хранимые в блоке указателя `*this` и в умном указателе `ptr`.

```
void unset() const;
```

Описание. Опустошить блок указателя `*this`.

Методы, сохраняющие состояние объекта и хранимые данные

```
Ref<Base> clone() const;
```

Описание. Вернуть клона указателя `*this`.

Методы доступа

```
bool empty() const;
```

Описание. Вернуть `true` ⇔ блок указателя `*this` пуст.

```
Base * get() const;
```

Описание. Вернуть основу блока указателя `*this`.

Требования. Блок указателя `*this` непуст.
Данные по возвращаемому адресу не должны явно или неявно уничтожаться никем, кроме хитрых указателей, разделяющих владение блоком указателя `*this`.

```
Base * get_copy() const;
```

Описание. Создать копию данных блока указателя `*this` и вернуть чистый указатель на неё.

```
bool nonempty() const;
```

Описание. Вернуть `true` ⇔ блок указателя `*this` непуст.

```
template<typename Value = Base>
Value & val() const;
```

Описание. Вернуть ссылку на данные блока указателя `*this`, трактуемые как данные типа `Value`.

Требования. Блок указателя `*this` непуст, хранимые данные имеют тип `Value`.

Статические методы

```
template<typename Value = Base, typename ... Args>
static Ref<Base> create(Args && ... args);
```

Описание. Вернуть хитрый указатель, связанный с обёрткой над `Value(args ...)`.

9.7 Типы функций работы с произвольными данными

Расположение

`type_abuse/dfun.hpp`

Общая ремарка

В дальнейшем описании указаны цели, с которыми вводятся синонимы типов функций. Эти цели можно и не соблюдать, они просто выражают то, что было у Автора в голове, когда он всё это придумывал.

Собственно типы

`typedef std::function<void(const DRef &, const DRef &)> Applier;`

Цель. В функциях такого типа данные блока второго аргумента применяются к данным блока первого аргумента — например, прибавляются, предоставляют условия изменения, дописываются в поля структуры и т.п. .

`typedef std::function<DRef()> Creator;`

Цель. Функции такого типа предназначены для динамического создания новых блоков данных (генерации новых данных).

`typedef std::function<void(const DRef &)> Modifier;`

Цель. Функция такого типа предназначена для пользовательского изменения данных блока аргумента.

10 Интерфейс модуля `regexp`

Все определения этого модуля располагаются в пространстве имён `pm::regexp`.

10.1 Общее описание

10.1.1 Что такое регулярное выражение

Регулярное выражение — это довольно известное понятие, с которым встречались, в частности, многие программисты, сталкивавшиеся с необходимостью разбора строк и потоков. В этой документации Автор постарался объяснить, что это такое и как с этим работать, достаточно чётко и доходчиво с расчётом на тех, кто не знает, что такое регулярное выражение, но знаком с математикой. Для лучше усваиваемости того, что реализовано в этом модуле, можно начать с описания регулярных выражений, используемого в математике.

Регулярное выражение r — это запись, определяемая следующей формой Бэкуса-Наура, в которой для устранения неоднозначности каждый элемент заключён в кавычки:

$$r \ ::= \text{“}\emptyset\text{”} \mid \text{“}\lambda\text{”} \mid \text{“}a\text{”} \mid \text{“}(r|r)\text{”} \mid \text{“}(r \cdot r)\text{”} \mid (r^*)$$

Здесь a — символ заранее известного конечного алфавита A, то есть конечного множества символов, а λ — пустая строка. Скобки можно опускать согласно приоритетам: наивысший приоритет у *, затем у $\cdot$, затем у $|$.

Компоненты записи (синтаксиса) регулярного выражения можно разбить на две части: базовые выражения (пустое множество $\emptyset$, пустое слово λ, символ a) и композициональные операции (альтернатива $|$, конкатенация $\cdot$, звезда Клини *). Синтаксиом регулярного выражения определяется общая структура “хороших” по мнению этого выражения строк.

Смысл (семантика) регулярного выражения r — это распознаваемый им язык $L(r)$, то есть множество принимаемых этим выражением строк. Выражением $\emptyset$ не принимаются никакие строки. Выражением λ принимается только пустая строка. Выражением a принимается только строка, состоящая из одного символа a. Выражением $r_1|r_2$ принимаются строки, принимаемые хотя бы одним из подвыражений r_1, r_2, и только они. Выражением $r_1 \cdot r_2$ принимаются строки, имеющие вид w_1w_2, где строка w_1 принимается подвыражением r_1, а строка r_2 — подвыражением r_2, и только они. Выражением r^* принимаются строки, имеющие вид $w_1w_2 \dots w_k$, где каждая строка w_i принимается подвыражением r, и только они.

Например, выражением $(0|1) \cdot (0|1)^*$ принимаются в точности все непустые строки из нолей и единиц (двоичные записи чисел, допускающие незначащие ноли).

В программировании регулярные выражения, как правило, отличаются более широким синтаксисом (больше базовых выражений, больше операций) и более узкой семантикой (принимаются не все строки, принимаемые в математическом смысле).

Регулярные выражения в программировании обычно применяются для следующей задачи: дана строка h; определить, принимается ли она, и если да, то совершить некоторые действия согласно структуре этой строки. Структура и соответствующий порядок действий определяются на основе разбора строки выражением — процесса последовательного чтения символов строки, согласно которому базовые выражения работают согласно заложенному в них алгоритму чтения и анализа строки, а операции в нужном порядке запускают разбор подстрок своими подвыражениями и делают выводы о приёме строки и результатах анализа. В результате успешного разбора некоторое прочитанное слово (не обязательно вся строка) принимается, а неуспешного — отвергается.

Так, например, обычно при разборе строки выражением

- $r_1|r_2$ запускается разбор строки выражением r_1, при неуспехе запускается разбор той же строки выражением r_2, и в случае успеха (хотя бы в одном подвыражении) принимается слово, принятое при этом успехе;
- $r_1 \cdot r_2$ запускается разбор строки выражением r_1, при успехе запускается разбор подстроки, начинающейся со следующего непрочитанного символа, выражением r_2, и в случае двух успехов принимается конкатенация первого и второго принятых слов;

- r^* раз за разом до первого неуспеха запускается разбор подстроки, начинающейся со следующего непрочитанного символа, выражением r, и принимается конкатенация всех слов, принятых при полученных успехах.

Можно пойти дальше и задаться вопросом, как быть, если нам нужно не только принимать слово, но и извлекать из точного устройства этого слова какую-либо информацию. Например, полезно было бы не просто принимать любую двоичную запись числа, но и извлекать само число, получая его в заданной переменной. Для этих целей каждый делает что ему заблагорассудится. Автор, например, предлагает расширить синтаксис регулярных выражений, чтобы для примера с двоичной записью можно было соорудить нечто такое:

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

Здесь $f()$ означает "сохрани в промежуточную переменную цифру, символ которой прочитался при разборе строки подвыражением-аргументом", а $g()$ — "помножь промежуточную переменную на 2 и прибавь к результату цифру, символ которой прочитался при разборе строки выражением-подаргументом". После разбора строки таким расширенным выражением в промежуточной переменной будет записано число, двоичная запись которого была принята. $f()$ и $g()$ из примера про извлечение числа из двоичной записи, а также подобные им конструкции, расширяющие разбор строки подвыражением-аргументом работой с генерируемыми данными, названы функциональными операциями.

10.1.2 Регулярные выражения в `regexp`

Синтаксис регулярных выражений, реализованных в `regexp`, включает в себя широкий набор базовых выражений, композициональных операций и функциональных операций, работающих согласно программистской семантике. Работать с регулярными выражениями можно, используя только нешаблонный класс `Regexp` и интерфейс создания выражений.

С помощью методов `match` класса `Regexp`[15] можно производить разбор трёх объектов: строки, стандартного потока ввода и возвратного потока (`stream::Stream`).

По итогам разбора пользователем по желанию могут быть получены принятая строка (в хитром указателе типа `type_abuse::BRef<std::string>`) и сгенерированные данные (в хитром указателе на данные произвольного типа `type_abuse::DRef`). Желание пользователя выражается посылкой в методы разбора специального аргумента — контекста разбора. Можно не задумываться над контекстом разбора и не посылать его в методы разбора — это будет означать, что ни строка, ни данные не требуются.[16]

Генерация изначальных данных — прочитанных символов и строк — производится базовыми выражениями. Композициональные операции предсказуемым образом запускают разбор своими подвыражениями, в том числе и обработку данных этими подвыражениями. Функциональные операции особым образом, зависящим от типа применяемой операции, изменяют данные, запуская в нужные моменты предоставленные пользователем функции работы с произвольными данными (`type_abuse::Applier`, `type_abuse::Creator`, `type_abuse::Modifier`) с нужными аргументами. Это означает, что пользователь может творить с данными, по большому счёту, всё что пожелает.

10.1.3 Контекст разбора

В методы разбора первым аргументом посылается разбираемый объект, а вторым, по желанию, — особый аргумент, контекст разбора (класс `Context`). В контексте содержатся четыре хитрых указателя:

1. Указатель строку, разобранную к текущему моменту.
   Имеет тип `type_abuse::BRef<std::string>`.
2. Указатель на данные, сгенерированные к текущему моменту.
   Имеет тип `type_abuse::DRef`.
3. Указатель на флаг активности строки.
   Имеет тип `type_abuse::BRef<bool>`.
4. Указатель на флаг активности данных.
   Имеет тип `type_abuse::BRef<bool>`.

Если флаг активности строки поднят, то при успешном разборе принятая строка добавляется в конец строки, содержащейся в соответствующем указателе к моменту начала разбора. В остальных случаях (флаг опущен или разбор неуспешен) разобранная строка не изменяется.

Если флаг активности данных поднят, то при успешном разборе генерируемые данные изменяются согласно семантике выражения. В остальных случаях (флаг опущен или разбор неуспешен) данные остаются без изменений.

Если контекст не посылается вторым аргументом, то на его месте подразумевается контекст по умолчанию (`default_context`). В этом контексте в начале работы программы содержатся пустая строка, пустые данные и опущенные флаги. Можно, но не рекомендуется изменять содержимое этого контекста в коде программы.

[15] Более точно, это методы базового класса `Literal`

[16] Более точно, будет послан контекст по умолчанию, в котором изъявлено желание не получать ни строку, ни данные. Этот контекст может изменяться пользователем, однако делать это не рекомендуется, иначе могут возникать ошибки из-за забытого очищения текущих разобранной строки и сгенерированных данных.

При копировании и сдвиге контекста, а также при применении специальных методов владение указателями разделяется между контекстами. Дублирование данных контекста и прочие хитрые действия над блоками данных производятся при помощи специальных методов контекста и методов хитрых указателей.

10.1.4 Подключение интерфейса использования готовых регулярных выражений

Интерфейсы создания всех базовых выражений подключаются заголовочным файлом `regexp/expr_basic.hpp`.

Интерфейсы применения всех композициональных операций подключаются заголовочным файлом `regexp/expr_composition.hpp`.

Интерфейсы применения всех функциональных операций подключаются заголовочным файлом `regexp/expr_function.hpp`.

Интерфейсы всех выражений и операций подключаются заголовочным файлом `regexp/expressions.hpp`

10.1.5 Написание своих регулярных выражений и операций

Пользователь может легко реализовать новое регулярное выражение. Для этого следует:

1. Реализовать класс, производный от класса `Literal`.
2. В реализованном классе перегрузить особый виртуальный метод `match_full` — метод полного разбора возвратного потока с учётом контекста.
3. Реализовать способ получения объекта класса `Regexp`, содержащего хитрый указатель на реализованный литерал, по аналогии с объектами и функциями интерфейса регулярных выражений данного модуля.

Чтобы понять, как это точно должно выглядеть, достаточно посмотреть в файлы `regexp/basic/let.hpp` и `regexp/basic/let.cpp`, в которых реализовано базовое выражение “заданный символ”.

10.1.6 Общие соглашения о разборе возвратного потока

При желании можно реализовать класс, позволяющий легко и просто получить регулярное выражение, делающее с возвратным потоком и контекстом что им заблагорассудится. Такая свобода действий чревата ошибками и непредсказуемостью результатов разбора. По этой причине существует набор соглашений о том, как должны быть устроены регулярные выражения.

Помимо назначения (базовое выражение, композициональная операция, функциональная операция), каждое регулярное выражение имеет тип: точное, либо неточное. Точное регулярное выражение пытается прочитать не более заданного количества символов из потока. Количество символов, читаемое из потока неточным регулярным выражением, невозможно точно специфицировать.

Регулярное выражение, являющееся результатом применения композициональной операции к подвыражениям, по определению считается неточным: читаемое количество символов зависит от такового в подвыражениях, и в процессе разбора символы читаются из потока и возвращаются в него в процессе вызовов подвыражений, за чем, в общем-то, сложно следить.

Регулярное выражение, являющееся результатом применения функциональной операции к подвыражению, настолько же точно, насколько и подвыражение: с точки зрения чтения из потока и возврата в него, функциональное выражение работает точно так же, как и его единственное подвыражение.

Базовые регулярные выражения могут быть как точными, так и неточными.

Каким бы ни было регулярное выражение, оно должно соблюдать следующие условия:

1. Метод полного разбора `match_full` возвращает `true` $\Leftrightarrow$ разбор завершился успехом.
2. Указатели на разобранную строку, генерируемые данные и флаги активации после разбора указывают на те же блоки, что и до разбора.
3. По итогам успешного разбора из потока читается ровно столько символов, какова длина принятой строки, и последовательность этих символов в точности образуют принятую строку. Если сверх этого из потока читались другие символы, то они возвращаются в поток, чтобы при дальнейшем чтении из потока всё выглядело так, будто они не читались.
4. Если в начале успешного разбора флаг активности строки был поднят, то по итогам разбора принятая строка добавляется в конец разобранной. В остальных случаях (разбор неуспешен, флаг был опущен) разобранные строки до и после разбора совпадают.
5. Если в начале успешного разбора флаг активности данных был поднят, то по итогам разбора данные изменяются согласно семантике. В остальных случаях (разбор неуспешен, флаг был опущен) данные до и после разбора совпадают.
6. Независимо от успешности разбора флаги активности строки и данных на момент начала разбора совпадают с флагами по итогам разбора.

В спецификации интерфейса регулярных выражений учтены все соглашения о разборе, и в связи с этим неявно подразумевается всё, кроме следующих явно указываемых деталей:

1. Тип базового выражения.
2. Количество читаемых символов для точного базового выражения.
3. Описание строки, принимаемой базовым выражением.

4. Способ вызова подвыражений и описание принимаемой строки для результатов применения композициональных операций.
5. Способ изменения генерируемых данных.

10.2 `class Context;`

Расположение

`regexp/context.hpp`

Описание

Класс контекста разбора.
Контекст содержит два поля (подконтекста):
1. Активационный строковый контекст `asc` для работы с разборанной строкой.
2. Активационный контекст данных `adc` для работы с генерируемыми данными.

С учётом внутренней структуры полей, контекст содержит четыре хитрых указателя:
1. Указатель на разобранную строку.
2. Указатель на флаг активности строки.
3. Указатель на генерируемые данные.
4. Указатель на флаг активности данных.

Конструкторы и присваивания по умолчанию, деструктор

Все в наличии и имеют стандартную семантику с поправкой на действия по умолчанию над хитрыми указателями.
`Context()` инициализирует контекст новыми блоками данных, содержащими пустую разобранную строку, пустой блок генерируемых данных и опущенные флаги.

Другие конструкторы

`Context(const context::AString & asc, const context::AData & adc);`
Описание. Направить хитрые указатели `*this` на блоки соответствующих указателей подконтекстов в аргументах.

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

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

`Context(context::AString & asc, context::AData && adc);`
Описание. Move-аналоги предыдущего конструктора.

Методы, изменяющие состояние объекта

`void detach();`
Описание. Перенаправить все четыре хитрых указателя `*this` на клонов связанных с ними блоков.

`void detach_receive(const Context & sdc);`
Описание. Перенаправить все четыре хитрых указателя `*this` на клонов блоков, связанных с соответствующими хитрыми указателями контекста `sdc`.

Методы, сохраняющие состояние объекта и изменяющие хранимые данные

`void receive(const Context & sdc) const;`
Описание. Перезаписать блоки всех указателей `*this` блоками соответствующих указателей контекста `sdc`.

Методы, сохраняющие состояние объекта и хранимые данные

`Context clone() const;`
Описание. Вернуть контекст, все хитрые указатели которого связаны с клонами блоков соответствующих указателей `*this`.

`void send(const Context & sdc) const;`
Описание. Перезаписать блоки всех указателей `sdc` блоками соответствующих указателей `*this`.

Методы доступа

`template<typename Value>`
`Value & data() const;`
Описание. Вернуть ссылку на генерируемые данные, трактуемые как объект типа `Value`.
Требования. Блок, связанный с указателем на генерируемые данные, непуст и хранит данные типа `Value`.

`bool is_active() const;`
Описание. Вернуть `true` ⇔ хотя бы один из двух флагов активности поднят.

`bool is_empty() const;`
Описание. Вернуть `true` ⇔ блок генерируемых данных пуст.

`bool is_inactive() const;`
Описание. Вернуть `true` ⇔ оба флага активности опущены.

`bool is_nonempty() const;`
Описание. Вернуть `true` ⇔ блок генерируемых данных непуст.

`std::string & string() const;`
Описание. Вернуть ссылку на разобранную строку.

Поля

`context::AString asc;`
Описание. Подконтекст, содержащий хитрый указатель на разобранную строку, хитрый указатель на флаг активности строки (активатор) и методы работы с ними.

`context::AData adc;`
Описание. Подконтекст, содержащий хитрый указатель на генерируемые данные, хитрый указатель на флаг активности данных (активатор) и методы работы с ними.

10.3 `class Literal;`

Расположение

`regexp/literal.hpp`

Описание

Это один из двух основных классов аппарата регулярных выражений (наряду с классом `Regexp`).
Литерал — это объект класса `Literal` или любого производного класса.
В этом классе содержатся методы разбора строки, входного потока и возвратного потока с учётом контекста и без него.
Методы разбора делегируют аргументы особенному виртуальному методу — методу полного разбора возвратного потока с учётом контекста (`match_full`).
Метод полного разбора класса `Literal` немедленно успешно завершает разбор, не затрагивая аргументы.

Конструкторы и присваивания по умолчанию, деструктор

Все в наличии, стандартная семантика.
Деструктор виртуальный.

Методы

`bool match(stream::Stream & s, Context & context = default_context) const;`
Описание. Вернуть результат работы метода `match_full` с теми же аргументами.

`bool match(std::istream & s, Context & context = default_context) const;`
Описание. Вернуть результат работы метода `match` с Потоком `stream::Stream(s)` и тем же вторым аргументом.

`bool match(const std::string & s, Context & context = default_context) const;`
Описание. Вернуть результат работы метода `match` с входным потоком — базой потока `std::stringstream(s)`, и тем же вторым аргументом.

`virtual bool match_full(stream::Stream & s, Context & context = default_context) const;`
Описание. Немедленно вернуть `true`.
Основное назначение метода — быть перегруженным в производном классе.

10.4 `class Regexp : public Literal;`

Расположение

`regexp/regexp.hpp`

Описание

Это один из двух основных классов аппарата регулярных выражений (наряду с классом `Literal`).
Наследуется весь интерфейс класса `Literal`, за исключением перегруженного метода полного разбора.
Метод полного разбора делегирует разбор объекту, хранящемуся в данных блока единственного поля — хитрого указателя на литерал.

Конструкторы и присваивания по умолчанию, деструктор

Все в наличии, стандартная семантика с поправкой на действия по умолчанию над хитрыми указателями.
`Regexp()` инициализирует регулярное выражение, делегирующее полный разбор копии тривиального литерала `lskip`.

Другие конструкторы

`Regexp(const type_abuse::CRef<Literal> & rl);`
Описание. Направить содержащийся хитрый указатель на блок, связанный с `rl`.

`Regexp(type_abuse::CRef<Literal> && rl);`
Описание. Move-аналог предыдущего конструктора.

Методы, сохраняющие состояние объекта

`match_full(stream::Stream & stream, Context & context) const override;`
Описание. Вызвать метод полного разбора литерала блока хранящегося указателя с теми же аргументами.
Вернуть результат этого вызова.

Поля

`type_abuse::CRef<Literal> rl;`
Описание. Хитрый указатель на литерал, которому `*this` делегирует полный разбор возвратного потока с учётом контекста.

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

Расположение

`regexp/context/activator.hpp`

Пространство имён описания

`pm::regexp::context`

Описание

Активатор — это объект класса `Activator`.
Активатор — это хитрый указатель на булево значение, трактуемое как флаг активности (чего бы то ни было, не связанного с активатором).
Флаг активности поднят, если имеет значение `true`, и опущен, если имеет значение `false`.
Поднять флаг активатора — это записать `true` в блок активатора.
Опустить флаг активатора — это записать `false` в блок активатора.

Наследование интерфейса

Наследуется весь интерфейс класса `type_abuse::BRef<bool>`.

Конструкторы и присваивания по умолчанию, деструктор

Конструктор без аргументов удалён.
Остальные конструкторы, присваивания и деструктор в наличии; семантика та же, что и в классе `type_abuse::BRef<bool>`.

Другие конструкторы

`Activator(bool act);`
Описание. Направить `*this` на обёртку над `act`.

Методы, изменяющие состояние объекта

`void detach_activate();`
Описание. Перенаправить `*this` на обёртку над `true`.

`void detach_deactivate();`
Описание. Перенаправить `*this` на обёртку над `false`.

Методы, сохраняющие состояние объекта и изменяющие хранимые данные

`void activate() const;`
Описание. Поднять флаг.

`void deactivate() const;`
Описание. Опустить флаг.

Методы, сохраняющие состояние объекта и хранимые данные

`Activator clone() const;`
Описание. Вернуть клона активатора `*this`.

10.6 `class context::AData;`

Расположение

`regexp/context/adata.hpp`

Пространство имён описания

`pm::regexp::context`

Описание

Подконтекст, используемый, в частности, в качестве поля контекста разбора.
В этом подконтексте содержатся активатор и хитрый указатель на произвольные данные.
Данные активны, если флаг активатора поднят, и неактивны, если флаг активатора опущен.

Конструкторы и присваивания по умолчанию, деструктор

Конструктор без аргументов удалён.
Остальные конструкторы, присваивания и деструктор все в наличии и имеют стандартную семантику с поправкой на действия по умолчанию над хитрыми указателями.

Другие конструкторы

`AData(bool act);`
Описание. Направить активатор на обёртку над `act`, а указатель на данные — на новый пустой блок.

`AData(const Activator & ac, const type_abuse::DRef & dc);`
Описание. Направить активатор и указатель на данные на блоки соответствующих аргументов.

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

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

`AData(Activator && ac, type_abuse::DRef && dc);`
Описание. Move-аналоги предыдущего конструктора.

Методы, изменяющие состояние объекта

`void detach();`
Описание. Открепить активатор и указатель на данные.

`void detach_receive(const AData & adc);`
Описание. Перенаправить активатор и указатель на данные на клонов блоков соответствующих указателей подконтекста `adc`.

```
template<typename Value, typename ... Args>
void detach_set_activate(Args && ... args));
```
Описание. Перенаправить активатор на обёртку над `true`, а указатель на данные — на обёртку над `Value(args ...)`.

```
template<typename Value, typename ... Args>
void detach_set_deactivate(Args && ... args));
```
Описание. Перенаправить активатор на обёртку над `false`, а указатель на данные — на обёртку над `Value(args ...)`.

`void detach_unset_activate();`
Описание. Перенаправить активатор на обёртку над `true`, а указатель на данные — на новый пустой блок.

`void detach_unset_deactivate();`
Описание. Перенаправить активатор на обёртку над `false`, а указатель на данные — на новый пустой блок.

Методы, сохраняющие состояние объекта и изменяющие хранимые данные

`void activate() const;`
Описание. Поднять флаг активатора.

`void deactivate() const;`
Описание. Опустить флаг активатора.

`void receive(const AData & adc) const;`
Описание. Перезаписать блоки активатора и данных соответствующими блоками подконтекста `adc`.

```
template<typename Value, typename ... Args>
void set_activate(Args && ... args) const;
```
Описание. Поднять флаг активатора и записать значение `Value(args ...)` в блок данных.

```
template<typename Value, typename ... Args>
void set_deactivate(Args && ... args) const;
```
Описание. Опустить флаг активатора и записать значение `Value(args ...)` в блок данных.

`void unset_activate() const;`
Описание. Поднять флаг активатора и опустошить блок данных.

`void unset_deactivate() const;`
Описание. Опустить флаг активатора и опустошить блок данных.

Методы, сохраняющие состояние объекта и хранимые данные

`AData clone() const;`
Описание. Вернуть подконтекст, хитрые указатели которого связаны с клонами блоков соответствующих указателей `*this`.

`void send(const AData & sdc) const;`
Описание. Перезаписать блок активатора и блок данных подконтекста `sdc` соответствующими блоками подконтекста `*this`.

Методы доступа

```
template<typename Value>
Value & data() const;
```
Описание. Вернуть ссылку на данные, трактуемые как объект типа `Value`.
Требования. Блок, связанный с указателем на данные, непуст и хранит данные типа `Value`.

`bool is_active() const;`
Описание. Вернуть `true` $\Leftrightarrow$ данные активны.

`bool is_empty() const;`
Описание. Вернуть `true` $\Leftrightarrow$ блок данных пуст.

`bool is_inactive() const;`

Описание. Вернуть `true` ⇔ данные неактивны.

`bool is_nonempty() const;`
Описание. Вернуть `true` ⇔ блок данных непуст.

Поля

`Activator ac;`
Описание. Активатор данных.

`type_abuse::DRef dc;`
Описание. Хитрый указатель на данные.

10.7 `class context::AString;`

Расположение

`regexp/context/astring.hpp`

Пространство имён описания

`pm::regexp::context`

Описание

Класс подконтекста, используемого, в частности, в качестве поля контекста разбора.
В этом подконтексте содержатся активатор и хитрый указатель на разобранную к данному моменту строку.
Строка активна, если флаг активатора поднят, и неактивна, если флаг активатора опущен.

Определение

`typedef AVal<std::string> AString`

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

Расположение

`regexp/context/aval.hpp`

Пространство имён описания

`pm::regexp::context`

Описание

Специализация `AVal<T>` содержит активатор и хитрый указатель на значение типа `T`.
Специализация этого класса предназначена для использования, в частности, в контексте разбора в качестве поля, хранящего всегда непустой хитрый указатель на значение фиксированного типа вместе с активатором — хитрым указателем на флаг активации этого значения.
Значение активно, если флаг активации поднят, и неактивно, если флаг активации опущен.

Требования к аргументу шаблона

В зависимости от используемых методов от типа `Value` может требоваться наличие конструктора без аргументов, конструктора присваивания, конструктора сдвига, стандартного оператора присваивания, стандартного оператора присваивания со сдвигом и других конструкторов, запрашиваемых пользователем.

Конструкторы и присваивания по умолчанию, деструктор

Конструктор без аргументов удалён.
Остальные конструкторы, присваивания и деструктор все в наличии и имеют стандартную семантику с поправкой на действия по умолчанию над хитрыми указателями.

Другие конструкторы

`AVal(bool act);`
Описание. Направить активатор на обёртку над `act`, а указатель на данные — на обёртку над `Value()`.

`AVal(bool act, const Value & v);`
Описание. Направить активатор на обёртку над `act`, а указатель на данные — на обёртку над копией объекта `v`.

`AVal(bool act, Value && v);`
Описание. Move-аналог предыдущего конструктора.

`AVal(const Activator & ac, const type_abuse::BRef<Value> & vc);`
Описание. Направить активатор и указатель на значение на блоки соответствующих аргументов.

`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);`
Описание. Move-аналоги предыдущего конструктора.

Методы, изменяющие состояние объекта

`void detach();`
Описание. Открепить активатор и указатель на значение.

`void detach_receive(const AVal<Value> & avc);`
Описание. Перенаправить активатор и указатель на значение на клонов блоков соответствующих указателей подконтекста `avc`.

```
template<typename ... Args>
void detach_set_activate(Args && ... args);
```
Описание. Перенаправить активатор на обёртку над `true`, а указатель на значение — на обёртку над `Value(args ...)`.

```
template<typename ... Args>
void detach_set_deactivate(Args && ... args);
```
Описание. Перенаправить активатор на обёртку над `false`, а указатель на значение — на обёртку над `Value(args ...)`.

Методы, сохраняющие состояние объекта и изменяющие хранимые данные

`void activate() const;`
Описание. Поднять флаг активатора.

`void deactivate() const;`
Описание. Опустить флаг активатора.

`void receive(const AVal<Value> & avc) const;`
Описание. Перезаписать блоки активатора и указателя на значение соответствующими блоками подконтекста `avc`.

```
template<typename ... Args>
void set_activate(Args && ... args) const;
```
Описание. Поднять флаг активатора и записать значение `Value(args ...)` в блок указателя на значение.

```
template<typename ... Args>
void set_deactivate(Args && ... args) const;
```
Описание. Опустить флаг активатора и записать значение `Value(args ...)` в блок указателя на значение.

Методы, сохраняющие состояние объекта и хранимые данные

`AData clone() const;`
Описание. Вернуть подконтекст, хитрые указатели которого связаны с клонами блоков соответствующих указателей `*this`.

`void send(const AVal<Value> & avc) const;`
Описание. Перезаписать блок активатора и блок указателя на значение подконтекста `avc` соответствующими блоками подконтекста `*this`.

Методы доступа

`bool is_active() const;`
Описание. Вернуть `true` ⇔ значение активно.

`bool is_inactive() const;`
Описание. Вернуть `true` ⇔ значение неактивно.

`Value & val() const;`
Описание. Вернуть ссылку на значение.

Поля

`Activator ac;`
Описание. Активатор значения.

`type_abuse::BRef<Value> vc;`
Описание. Хитрый указатель на хранимое значение.

10.9 `class context::String;`

Расположение

`regexp/context/string.hpp`

Пространство имён описания

`pm::regexp::context`

Описание

Сокращение для хитрого указателя, немного укорачивающее запись типов при работе с контекстом разбора и его подконтекстами.

Определение

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

10.10 Интерфейс базовых регулярных выражений

`const Regexp anylet;`
Расположение. `regexp/basic/anylet.hpp`
Тип. Точное, 1 символ.
Приём строки. Принимается любой успешно прочитанный символ.
Изменение данных. В данные записывается прочитанный символ.

`Regexp anyletbut(char bad_char);`
Расположение. `regexp/basic/anyletbut.hpp`
Тип. Точное, 1 символ.
Приём строки. Принимается любой успешно прочитанный символ, кроме `bad_char`.
Изменение данных. В данные записывается прочитанный символ.

`Regexp anyletbutset(const std::set<char> & bad_chars);`
Расположение. `regexp/basic/anyletbutset.hpp`
Тип. Точное, 1 символ.
Приём строки. Принимается любой успешно прочитанный символ, кроме символов множества `bad_chars`.
Изменение данных. В данные записывается прочитанный символ.

`Regexp anyletbutset(std::set<char> && bad_chars);`
Расположение. `regexp/basic/anyletbutset.hpp`
Описание. Move-аналог предыдущей функции

`Regexp anystr(size_t size);`
Расположение. `regexp/basic/anystr.hpp`
Тип. Точное, `size` символов.
Приём строки. Принимается любая успешно прочитанная строка длины `size`.
Изменение данных. В данные записывается прочитанная строка.

`Regexp enclose(const std::string & opening_string, const std::string & closing_string);`
Расположение. `regexp/basic/enclose.hpp`

Тип. Неточное.

Приём строки. Принимается кратчайшая строка, имеющая следующий вид: `opening_string`, затем произвольная строка (в том числе пустая), затем `closing_string`.

Изменение данных. В данные записывается прочитанная строка.

Комментарий. Данное выражение включено в список базовых, так как невероятно часто именно так определяется синтаксис комментариев в языках программирования.
Например, в языке `C++` существуют два типа комментариев, описываемых такими регулярными выражениями:

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);`

Расположение. `regexp/basic/enclose.hpp`

Описание. Move-аналоги предыдущей функции.

`const Regexp end;`

Расположение. `regexp/basic/end.hpp`

Тип. Точное, 1 символ.

Приём строки. Из потока читается один символ, и он же немедленно возвращается в поток.
Принимается пустая строка, и только в том случае, если чтение символа оказалось неуспешным.

Изменение данных. Данные не изменяются.

`const Regexp fail;`

Расположение. `regexp/basic/fail.hpp`

Тип. Точное, 0 символов.

Приём строки. Разбор всегда неуспешен.

`Regexp let(char good_char);`

Расположение. `regexp/basic/let.hpp`

Тип. Точное, 1 символ.

Приём строки. Принимается успешно прочитанный символ, равный `good_char`.

Изменение данных. В данные записывается символ `good_char`.

`Regexp letset(const std::set<char> & good_chars);`

Расположение. `regexp/basic/letset.hpp`

Тип. Точное, 1 символ.

Приём строки. Принимается любой успешно прочитанный символ, принадлежащий множеству `good_chars`.

Изменение данных. В данные записывается прочитанный символ.

`Regexp letset(std::set<char> && good_chars);`

Расположение. `regexp/basic/letset.hpp`

Описание. Move-аналог предыдущей функции.

`Regexp range(char bottom_char, char top_char);`

Расположение. `regexp/basic/range.hpp`

Тип. Точное, 1 символ.

Приём строки. Принимается любой успешно прочитанный символ `c`, такой что `bottom_char <= c <= top_char`.

Изменение данных. В данные записывается прочитанный символ.

`const Regexp skip;`

Расположение. `regexp/regexp.hpp`

Тип. Точное, 0 символов.

Приём строки. Разбор всегда успешен, принимается пустая строка.

Изменение данных. Данные не изменяются.

`Regexp str(const std::string & good_string);`

Расположение. `regexp/basic/str.hpp`

Тип. Точное, столько символов, какова длина строки `good_string`.

Приём строки. Принимается успешно прочитанная строка, посимвольно равная `good_string`.

Изменение данных. В данные записывается прочитанная строка.

`Regexp str(std::string && good_string);`

Расположение. `regexp/basic/str.hpp`

Описание. Move-аналог предыдущей функции.

10.11 Интерфейс композициональных операций

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

Расположение. `regexp/composition/alt.hpp`

Приём строки. Запускается разбор подвыражения `r1`.
В случае успеха разбор выражения успешен.
В случае неуспеха запускается разбор подвыражения `r2`, и успешность разбора выражения — это успешность разбора `r2`.
Принимается строка, принятая подвыражением, разбор которого окончился успехом.

Изменение данных. Данные преобразуются согласно подвыражению, разбор которого окончился успехом.

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

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

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

Расположение. `regexp/composition/alt.hpp`

Описание. Move-аналоги предыдущей функции.

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

Расположение. `regexp/composition/concat.hpp`

Приём строки. Запускается разбор подвыражения `r1`.
В случае неуспеха разбор выражения неуспешен.
В случае успеха запускается разбор подвыражения `r2`, успешность разбора выражения — это успешность разбора `r2`.
Принимается строка `s1 + s2`, где `s1` — строка, принятая подвыражением `r1`, а `s2` — строка, принятая подвыражением `r2`.

Изменение данных. Данные преобразуются согласно подвыражению `r1`, и затем согласно подвыражению `r2`.

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

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

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

Расположение. `regexp/composition/concat.hpp`

Описание. Move-аналоги предыдущей функции.

`Regexp operator !(const Regexp & r;`

Расположение. `regexp/composition/opt.hpp`

Приём строки. Запускается разбор подвыражения `r`.
В случае успеха принимается строка, принятая `r`.
В случае неуспеха принимается пустая строка.
Разбор выражения всегда успешен.

Изменение данных. В случае неуспешного разбора подвыражения данные не изменяются.
В случае успешного разбора подвыражения данные преобразуются согласно подвыражению.

`Regexp operator !(Regexp && r;`

Расположение. `regexp/composition/opt.hpp`

Описание. Move-аналог предыдущей функции.

`Regexp operator *(const Regexp & r);`

Расположение. `regexp/composition/star.hpp`

Приём строки. Разбор подвыражения `r` запускается раз за разом, пока не будет получен неуспех.
Принимается строка `s1 + s2 + ... + sk`, где `k` — количество успешных разборов подвыражения, и `si` — строка, принятая подвыражением на `i`-м запуске.

Изменение данных. Данные последовательно преобразуются согласно каждому успешному разбору подвыражения.

`Regexp operator *(Regexp && r);`

Расположение. `regexp/composition/star.hpp`

Описание. Move-аналог предыдущей функции.

10.12 Интерфейс функциональных операций

Общее описание

Выражение, полученное в результате применения функциональной операции, работает так. Единственное подвыражение запускается с контекстом, отличающимся от текущего заменой хитрого указателя на генерируемые данные другим указателем. После этого производится действие над исходными данными и данными, полученными по результатам разбора подвыражения (в некоторых случаях никаких действий не происходит).

`main` — указатель на данные в исходном контексте.

`sub` — указатель на данные в контексте, с которым запускается подвыражение.

“`sub = main`” означает, что указатель `sub` не заменяется, то есть указывает на тот же блок, что и `main`.

Название каждой из функциональных операций состоит из двух букв. Первая буква обозначает вид данных в контексте подвыражения. Вторая буква обозначает наличие и вид действий после разбора подвыражения.

Буква “`a`” на второй позиции означает клонирование исходных генерируемых данных, применение к ним данных, полученных после разбора подвыражения (при помощи функции типа `type_abuse::Applier`), и выдачу результата применения в качестве итоговых генерируемых данных.

Буква “`c`” означает генерацию данных согласно функции по содержащемуся указателю типа `type_abuse::Creator`.

Буква “`m`” означает модификацию данных согласно функции по содержащемуся указателю типа `type_abuse::Modifier`.

Буква “`n`” на первой позиции означает отключение генерации данных в подвыражении.

Буква “`x`” означает, что генерируемые данные не затрагиваются.

Собственно операции

`Regexp ca(const type_abuse::Creator & cr, const Regexp & r, const type_abuse::Applier & ap);`
Расположение. `regexp/function/ca.hpp`
Изменение данных. `sub = cr()`.
После разбора выполняется `ap(main, sub)`.

`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);`
Расположение. `regexp/function/ca.hpp`
Описание. Move-аналоги предыдущей функции.

`Regexp cm(const type_abuse::Creator & cr, const Regexp & r, const type_abuse::Modifier & md);`
Расположение. `regexp/function/cm.hpp`
Изменение данных. `sub = cr()`.
После разбора выполняется `md(sub)`, и данные блока указателя `main` подменяются данными блока указателя `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);`
Расположение. `regexp/function/cm.hpp`
Описание. Move-аналоги предыдущей функции.

`Regexp cx(const type_abuse::Creator & cr, const Regexp & r);`
Расположение. `regexp/function/cx.hpp`
Изменение данных. `sub = cr()`.
После разбора данные блока указателя `main` подменяются данными блока указателя `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);`
Расположение. `regexp/function/cx.hpp`
Описание. Move-аналоги предыдущей функции.

`Regexp ma(const type_abuse::Modifier & md, const Regexp & r, const type_abuse::Applier & ap);`
Расположение. `regexp/function/ma.hpp`
Изменение данных. `sub = md(main.clone())`.
После разбора выполняется `ap(main, sub)`.

`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);`

Расположение. `regexp/function/ma.hpp`

Описание. Move-аналоги предыдущей функции.

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

Расположение. `regexp/function/mm.hpp`

Изменение данных. `sub = md1(main.clone())`.
После разбора выполняется `md2(sub)`, и данные блока указателя `main` подменяются данными блока указателя `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);`

Расположение. `regexp/function/mm.hpp`

Описание. Move-аналоги предыдущей функции.

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

Расположение. `regexp/function/mx.hpp`

Изменение данных. `sub = md(main.clone())`.
После разбора данные блока указателя `main` подменяются данными блока указателя `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);`

Расположение. `regexp/function/mx.hpp`

Описание. Move-аналоги предыдущей функции.

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

Расположение. `regexp/function/nc.hpp`

Изменение данных. `sub` указывает на новый пустой блок. Активатор данных контекста разбора подвыражения указывает на новый блок, содержащий опущенный флаг.
После разбора данные блока указателя `main` подменяются данными блока указателя `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);`

Расположение. `regexp/function/nc.hpp`

Описание. Move-аналоги предыдущей функции.

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

Расположение. `regexp/function/nm.hpp`

Изменение данных. `sub` указывает на новый пустой блок. Активатор данных контекста разбора подвыражения указывает на новый блок, содержащий опущенный флаг.
После разбора выполняется `md(main)`.

`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);`

Расположение. `regexp/function/nm.hpp`

Описание. Move-аналоги предыдущей функции.

`Regexp nx(const Regexp & r);`

Расположение. `regexp/function/nx.hpp`

Изменение данных. `sub` указывает на новый пустой блок. Активатор данных контекста разбора подвыражения указывает на новый блок, содержащий опущенный флаг.

`Regexp nx(Regexp && r);`
Расположение. `regexp/function/nx.hpp`
Описание. Move-аналог предыдущей функции.

`Regexp xa(const Regexp & r, const type_abuse::Applier & ap);`
Расположение. `regexp/function/xa.hpp`
Изменение данных. `sub = main.clone()`.
После разбора выполняется `ap(main, sub)`.

`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);`
Расположение. `regexp/function/xa.hpp`
Описание. Move-аналоги предыдущей функции.

`Regexp xm(const Regexp & r, const type_abuse::Modifier & md);`
Расположение. `regexp/function/xm.hpp`
Изменение данных. `sub = main`.
После разбора выполняется `md(main)`.

`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);`
Расположение. `regexp/function/xm.hpp`
Описание. Move-аналоги предыдущей функции.

11 Интерфейс модуля `dataset`

Все определения этого модуля располагаются в пространстве имён `pm::dataset`.

11.1 Общее описание

Автор рекомендует расценивать этот модуль как составную часть модуля `grammar`, которая вдруг стала самостоятельной. Но если вдруг кому-то модуль будет интересен сам по себе, то и хорошо.

Интерфейс этого модуля можно концептуально разбить на следующие части:

1. Совокупность данных. В этой части располагается основной класс `Set`, обобщающий интерфейс работы с произвольными данными (`type_abuse::DRef`) на интерфейс работы с наборами произвольных данных постоянного размера.
2. Функции над совокупностью данных. В этой части располагаются:
   - класс `FBase`, наследование которого означает “объекты этого класса можно применить к совокупности данных, чтобы с ней что-то произошло”;
   - класс `Function`, производный от `FBase` и хранящий хитрый указатель типа `type_abuse::CRef<FBase>`, — с помощью этого класса можно единообразно и нешаблонно работать с объектами, производными от `FBase` (то есть функциями над совокупностью данных);
   - интерфейс работы с некоторыми видами функций над совокупностью данных; можно подключить этот интерфейс целиком при помощи заголовочного файла `dataset/functions.hpp`.
3. Синтаксическая часть. Располагается в подпространстве имён `syntax`. Здесь описывается интерфейс удобного создания совокупностей данных и функций над ними, основанный на
   - синтаксическом описании совокупностей и функций, в котором каждый блок данных совокупности описывается строкой — именем, и
   - интерфейс получения совокупностей данных и функций над ними по их синтаксическому описанию.

11.1.1 Особенности трансляции

Интерфейс трансляции предоставляется несколькими глобальными функциями — функциями трансляции — с общим названием `syntax::interpret`. Есть два типа трансляции:

1. Получение совокупности данных по синтаксическому описанию.
2. Получение функции над совокупностью данных по синтаксическому описанию функции и синтаксическому описанию совокупности данных.

Синтаксическое описание совокупности данных — это объект класса `syntax::Set`. Синтаксическое описание функции (над совокупностью данных) — это объект класса `syntax::Function`.

К моменту вызова функции `interpret` все имена для хитрых указателей на произвольные данные должны быть придуманы (уникальное имя — уникальный указатель) и специальными методами внесены в синтаксическое описание совокупности данных. Для упрощения внесения имён класс `syntax::Set` содержит метод, вносящий все имена, содержащиеся в синтаксическом описании функции. Имя — это произвольная строка.

Совокупность данных возвращается функцией трансляции, принимающей синтаксическое описание совокупности данных. Функция над совокупностью данных возвращается методом трансляции принимающим два аргумента: синтаксическое описание функции, и синтаксическое описание совокупности данных, которым описывается трансляция имён в смещения в векторе совокупности данных.

Для пользователя абсолютно неважно, как устроен класс `syntax::Function`, важно только то, как получать объекты этого класса для последующей передачи их в функции трансляции. В связи с этим класс `syntax::Function` не описывается в документации, хотя объекты этого класса и возвращаются интерфейсом создания синтаксических описаний функций.

11.2 `class FBase;`

Расположение

`dataset/fbase.hpp`

Описание

Базовый класс всех функций над совокупностью данных (объектом типа `Set`).

Конструкторы и присваивания по умолчанию, деструктор

Все в наличии, стандартная семантика.
Деструктор виртуальный.

Методы

`virtual void apply(const Set &) const;`
Описание. Немедленно завершает работу, не затрагивая совокупность данных.
Назначение этого метода — быть перегруженным в производных классах.

11.3 `class Function : public FBase;`

Расположение

`dataset/function.hpp`

Описание

Класс, содержащий хитрый указатель на объект типа, производного от `FBase`.
Перегруженный метод `apply` делегирует изменение совокупности данных объекту, хранящемуся по указателю.
Назначение этого класса — организация нешаблонной единообразной работы с функциями преобразования совокупностей данных.

Конструкторы и присваивания по умолчанию, деструктор

Все присутствуют, стандартная семантика.

Другие конструкторы

`Function(const type_abuse::CRef<FBase> & pf);`
Описание. Направить хранящийся указатель на блок данных указателя `pf`.
Требования. Если будет вызываться метод `this->apply`, то блок указателя `pf` должен быть непуст.

`Function(type_abuse::CRef<FBase> && pf);`
Описание. Move-аналог предыдущего конструктора

Методы

`void apply(const Set & ds) const override;`
Описание. Метод применения `*this` к совокупности данных.
Вызывает метод `apply` с тем же аргументом у данных блока хранящегося хитрого указателя.

11.4 class Set;

Расположение

dataset/set.hpp

Описание

Класс совокупности данных.
Это вектор указателей на данные произвольного типа (std::vector<type_abuse::DRef>), обёрнутый методами, с одной стороны ограничивающими работу с вектором (например, запрещающие изменять длину), а с другой — позволяющие удобно работать содержащимися указателями.

Конструкторы и присваивания по умолчанию, деструктор

Конструктор без аргументов удалён.
Остальные конструкторы, присваивания и деструктор все в наличии, стандартная семантика с поправкой на действия по умолчанию над хитрыми указателями.

Другие конструкторы

`Set(std::initializer_list<type_abuse::DRef> l);`
Описание. Инициализировать вектор данных длины, равной длине списка `l`, и направить указатели вектора на соответствующие блоки элементов списка.

`Set(const std::vector<type_abuse::DRef> & v);`
Описание. Инициализировать вектор данных длины, равной длине вектора `v`, и направить указатели вектора на соответствующие блоки элементов вектора.

`Set(std::vector<type_abuse::DRef> && v);`
Описание. Move-аналог предыдущего конструктора

`Set(size_t size);`
Описание. Инициализировать вектор данных длины `size` и направить каждый указатель на отдельный новый пустой блок данных.

Другие операторы

`type_abuse::DRef & operator [](size_t i);`
Описание. Вернуть указатель по смещению `i` в векторе данных.
Требования. Вектор имеет длину не менее `i+1`.

`const type_abuse::DRef & operator [](size_t i) const;`
Описание. Const-аналог предыдущего оператора.

Методы, изменяющие состояние объекта

`void detach();`
Описание. Открепить каждый указатель вектора данных.

`void detach_receive(const Set & set);`
Описание. Перенаправить каждый указатель вектора данных `*this` на клона блока указателя с равным смещением в векторе данных совокупности `set`.
Требования. Длина вектора данных совокупности `set` больше или равна длине вектора данных совокупности `*this`.

`void detach_unset();`
Описание. Перенаправить каждый указатель вектора данных на новый уникальный пустой блок.

Методы, сохраняющие состояние объекта и изменяющие хранимые данные

`void receive(const Set & set) const;`
Описание. Перезаписать блок каждого указателя вектора данных `*this` блоком указателя с равным смещением в векторе данных совокупности `set`.
Требования. Длина вектора данных совокупности `set` больше или равна длине вектора данных совокупности `*this`.

`void unset() const;`
Описание. Опустошить блок каждого указателя в векторе данных.

Методы, сохраняющие состояние объекта и хранимые данные

`Set clone() const;`
Описание. Вернуть совокупность данных с вектором такой же длины, как у `*this`, такую что каждый элемент этого вектора связан с клоном блока с равным смещением в векторе данных `*this`.

`void send(const Set & set) const;`
Описание. Перезаписать блоки стольких первых указателей вектора данных `set`, каков размер вектора данных `*this`, блоками указателей с равным смещением в векторе данных совокупности `*this`.
Требования. Длина вектора данных совокупности `set` больше или равна длине вектора данных совокупности `*this`.

Методы доступа, кроме итерации

`type_abuse::DRef & at(size_t i);`
Описание. Вернуть указатель по смещению `i` в векторе данных.
Требования. Вектор имеет длину не менее `i+1`.

`const type_abuse::DRef & at(size_t i) const;`
Описание. Const-аналог предыдущего метода.

`size_t size() const;`
Описание. Вернуть размер вектора данных.

Методы итерации

`std::vector<type_abuse::DRef>::iterator begin();`
Описание. Вернуть итератор на начало вектора данных.

`std::vector<type_abuse::DRef>::const_iterator begin() const;`
Описание. Const-аналог предыдущего метода.

`std::vector<type_abuse::DRef>::iterator end();`
Описание. Вернуть итератор на конец вектора данных.

`std::vector<type_abuse::DRef>::const_iterator end() const;`
Описание. Const-аналог предыдущего метода.

11.5 Интерфейс создания функций над совокупностью данных

Общее описание

В дальнейших описаниях приведено описание действия, производимого на совокупность данных `ds` вызовом `apply(ds)` метода возвращаемой функции.

Собственно интерфейс

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

`Function applier(size_t t, size_t s, type_abuse::Applier && f);`
Расположение. `dataset/functions/applier.hpp`
Описание. Move-аналог предыдущей функции.

`Function cloner(size_t t, size_t s);`
Расположение. `dataset/functions/cloner.hpp`
Действие. Перезаписать блок указателя `ds[t]` блоком указателя `ds[s]`.

`Function composition(const std::list<Function> & funcs);`
Расположение. `dataset/functions/composition.hpp`
Действие. Последовательно выполнить действия функций списка `funcs`.

`Function composition(std::list<Function> && funcs);`
Расположение. `dataset/functions/composition.hpp`
Описание. Move-аналог предыдущей функции.

`Function creator(size_t t, const type_abuse::Creator & f);`
Расположение. `dataset/functions/creator.hpp`
Действие. Перезаписать блок указателя `ds[t]` блоком указателя `f()`.

`Function creator(size_t t, type_abuse::Creator && f);`

Расположение. `dataset/functions/creator.hpp`
Описание. Move-аналог предыдущей функции.

`Function flusher(size_t t, const type_abuse::DRef & d);`
Расположение. `dataset/functions/flusher.hpp`
Действие. Перезаписать блок указателя `ds[t]` блоком указателя `d`.

`Function applier(size_t t, type_abuse::DRef && d);`
Расположение. `dataset/functions/flusher.hpp`
Описание. Move-аналог предыдущей функции.

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

`Function modifier(size_t t, type_abuse::Modifier && f);`
Расположение. `dataset/functions/modifier.hpp`
Описание. Move-аналог предыдущей функции.

11.6 `class syntax::Set;`

Расположение

`dataset/syntax/set.hpp`

Пространство имён описания

`pm::dataset::syntax`

Описание

Класс синтаксического описания совокупности данных.
Описание подаётся в качестве аргумента функциям трансляции для получения совокупности данных и функции над совокупностью данных.
В каждый момент жизни описание содержит некоторое множество имён (произвольных строк).

Конструкторы и присваивания по умолчанию, деструктор

Все в наличии, стандартная семантика.
`Set()` инициализирует описание, содержащее пустое множество имён.

Методы, изменяющие состояние объекта

`size_t add(const std::string & name);`
Описание. Добавить имя в множество имён, если оно ещё там не содержится.
В любом случае вернуть смещение в векторе совокупности данных, в которое будет транслироваться `name`, до тех пор пока не будет вызван метод, опустошающий множество имён.

`size_t add(std::string && name);`
Описание. Move-аналог предыдущего метода.

`void add(const Function & fun);`
Описание. Добавить все имена, используемые в синтаксическом описании функции над совокупностью данных `fun`, в множество имён.

`void add(Function && fun);`
Описание. Move-аналог предыдущего метода.

`void clear();`
Описание. Опустошить множество имён.

Методы доступа

`bool has_name(const std::string & name) const;`
Описание. Вернуть `true` ⇔ имя `name` содержится в множестве имён.

`size_t index_of(const std::string & name) const;`
Описание. Вернуть смещение в векторе совокупности данных, в которое будет транслироваться `name`, до тех пор пока не будет вызван метод, опустошающий множество имён.
Требования. Имя `name` содержится в множестве имён.

`const utils::Numeration<std::string> & names() const;`

Описание. Вернуть нумерацию, описывающую отображение множества всех содержащихся имён в смещения в векторе совокупности данных, в которые будут транслироваться эти имена, пока не будет вызван метод, опустошающий множество имён.

`size_t size() const;`
Описание. Вернуть размер множества имён.

11.7 Интерфейс трансляции

Расположение

`dataset/syntax/interpreter.hpp`

Пространство имён описания

`pm::dataset::syntax`

Собственно интерфейс

`dataset::Set interpret(const Set & set);`
Описание. Вернуть результат трансляции синтаксического описания совокупности данных `set` в совокупность данных.
Длина вектора данных результата равна количеству уникальных имён в описании `set`.
Каждое имя в описании соответствует уникальному допустимому смещению в векторе данных.
Каждый хитрый указатель вектора данных результата связан с уникальным новым пустым блоком.

`dataset::Function interpret(const Function & fun, const Set & set);`
Описание. Вернуть функцию над совокупностью данных, соответствующую синтаксическому описанию `fun` в контексте описания совокупности данных `set` согласно документации интерфейса получения синтаксических описаний функций над совокупностью данных.
Требования. Каждое имя, используемое в `fun`, должно содержаться в описании `set`.

`dataset::Function interpret(Function && fun, const Set & set);`
Описание. Move-аналог предыдущей функции.

11.8 Интерфейс создания синтаксического описания функций над совокупностью данных

Расположение

`dataset/syntax/function.hpp`

Пространство имён описания

`pm::dataset::syntax`

Общее описание

Интерфейс позволяет получать объекты типа `Function`, подаваемые затем в синтаксическое описание совокупности данных и в функции трансляции. В описании интерфейса говорится, в какую функцию над совокупностью данных транслируется получаемое синтаксическое описание функции в контексте готового синтаксического описания совокупности данных. Чтобы сказать это покороче, используется сокращение `<name>`: смещение хитрого указателя на произвольные данные, которому соответствует имя `name` при трансляции синтаксического описания совокупности данных.

Собственно интерфейс

`Function trivial_function;`
Соответствие. Тривиальная функция `dataset::Function()`.

`Function applier(const std::string & target, const std::string & source, const type_abuse::Applier & fun);`
Соответствие. `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);`
Описание. Move-аналоги предыдущей функции.

`Function cloner(const std::string & target, const std::string & source);`
Соответствие. `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);`
Описание. Move-аналоги предыдущей функции.

`Function composition(const std::list<Function> & funcs);`
Соответствие. `dataset::composition(<funcs>)`, где `<funcs>` — список результатов трансляции синтаксических описаний списка `funcs`.

`Function composition(std::list<Function> && funcs);`
Описание. Move-аналог предыдущей функции.

`Function creator(const std::string & target, const type_abuse::Creator & fun);`
Соответствие. `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);`
Описание. Move-аналоги предыдущей функции.

`Function flusher(const std::string & target, const type_abuse::DRef & data);`
Соответствие. `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);`
Описание. Move-аналоги предыдущей функции.

`Function modifier(const std::string & target, const type_abuse::Modifier & fun);`
Соответствие. `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);`
Описание. Move-аналоги предыдущей функции.

12 Интерфейс модуля grammar

Все определения этого модуля располагаются в пространстве имён `pm::grammar`.

12.1 Общее описание

12.1.1 Что такое грамматика

Грамматика — это довольно известное понятие, с которым встречались, в частности, многие программисты, сталкивавшиеся с необходимостью разбора строк и потоков. Это понятие устроено более хитро, чем регулярные выражения, и применяется для разбора сложных структурированных текстов — например (но не только), текстов программ на языках программирования. В этой документации Автор постарался объяснить, что это такое и как с этим работать, достаточно чётко и доходчиво с расчётом на тех, кто не знает, что такое грамматика, но знаком с математикой. Для лучше усваиваемости того, что реализовано в этом модуле, можно начать с описания грамматик, используемого в математике.

В модуле реализовано нечто похожее на контекстно-свободные грамматики. В свете этого факта следует иметь в виду, что во всех определениях, пояснениях и тому подобном словом “грамматика” обозначается именно контекстно-свободная грамматика (или нечто схожее, если говорить о том, что реализовано в данном модуле).

Грамматика G над конечным алфавитом $\mathcal{A}$ (конечным множеством символов; алфавитом терминалов) состоит из:

1. Конечного множества нетерминалов $\mathcal{N}$.
2. Выделенного начального нетерминала S, $S \in \mathcal{N}$.
3. Конечного множества правил $\mathcal{R}$. Каждое правило имеет вид $A \to BODY$, где A — левая часть правила, являющаяся нетерминалом, и $BODY$ — правая часть правила, являющаяся последовательностью нетерминалов и терминалов.

Значение (семантика) грамматики G, как значение регулярного выражения, — это язык $L(G)$, распознаваемый этой грамматикой, то есть множество принимаемых строк. Это множество можно определить так. Возьмём строку, состоящую из одного начального нетерминала, и начнём её преобразовывать шаг за шагом, каждый раз получая некоторую строку из терминалов и нетерминалов. Шаг преобразования выглядит так:

- выберем один (любой) нетерминал в текущей строке и одно (любое) правило, в левой части которого стоит этот нетерминал;
- заменим выбранный нетерминал в строке на правую часть этого правила.

Такой шаг преобразования называется развёрткой нетерминала согласно правилу. Язык грамматики состоит в точности из всех слов, которые можно получить, производя такие развёртки.

Типичный пример нетривиальной грамматики — это грамматика сбалансированных скобок, которая в простом случае может содержать один нетерминал S и два правила: $S \to (S)S$ и $S \to \lambda$. Здесь λ — пустая строка. Нетрудно видеть, что язык, распознаваемый такой грамматикой, состоит в точности из всех строк, состоящих только из символов открывающей и закрывающей скобок и при этом сбалансированных по скобкам.

Грамматики, используемые в программировании, как и в случае регулярных выражений, часто отличаются от математических более широким синтаксисом и более узкой семантикой. С одной стороны, добавляются новые типы правил и свёрток. С другой стороны, не все строки, принимаемые грамматикой в математическом смысле, принимаются ей в программистском.

Грамматики в программировании используются обычно для решения следующей задачи: дана строка; проверить, принимается ли строка грамматикой, и если да, то произвести некоторые действия, определяемые точной структурой этой строки. Решение этой задачи, в процессе которого строятся промежуточные структуры, как правило основанные на строках из терминалов и нетерминалов, и просматриваются (читаются) символы строки, называется разбором строки. В связи с устройством стандартного потока ввода и вообще понятия “поток”, программистам наиболее близок так называемый левый разбор, в процессе которого символы строки читаются по порядку от первого (левого) к последнему (правому). Далее под разбором понимается левый разбор, во время которого также разрешено смещаться обратно влево в строке к уже прочитанным символам (или возвращать символы обратно в поток).

В данном модуле используется один особенный способ разбора. Для тех, кто в курсе, — это метод рекурсивного спуска с возвратом. Выглядит он примерно так. Упорядочим все правила грамматики. Вызовем процедуру “разобрать строку начальным нетерминалом”.

Для каждого терминала определена подобная процедура, и она заключается в следующем:

1. Вызовем процедуру “разобрать строку первым правилом, в левой части которого стоит этот нетерминал”.
2. Если вызванная процедура успешно завершилась, то успешно завершим текущую процедуру, иначе вызовем процедуру “разобрать строку вторым таким правилом”.
3. ...
4. Если правила, в левых частях которых стоит этот нетерминал, кончились и все завершились неуспехом, то неуспешно завершим процедуру.

Процедура “разобрать строку правилом” работает так. Будем двигаться от первого символа правила к последнему и производить следующие действия:

1. Если текущий символ правила — терминал, то попытаемся прочитать следующий символ разбираемой строки. Если прочитанный символ совпал с терминалом, то продолжим движение, иначе неуспешно завершим процедуру.
2. Если текущий символ правила — нетерминал, то вызовем процедуру “разобрать строку этим нетерминалом”. Если вызванная процедура завершилась успехом, то продолжим движение, иначе неуспешно завершим текущую процедуру.
3. Если движение завершено (символы правила кончились), то успешно завершим процедуру.
4. При неуспешном завершении сместимся влево в строке так, как будто текущая процедура не вызывалась.

Как можно видеть, такой рекурсивный спуск принимает далеко не все слова, принимаемые грамматикой в математическом смысле. Но всё-таки такой разбор достаточно полезен и нетривиален.

Можно пойти дальше и задаться вопросом, как в процессе разбора совершать полезные действия, основанные на точной структуре разбираемой строки и отличные от чтения символов и вызова функций разбора. Например,[17] при разборе строки грамматикой сбалансированных скобок можно захотеть посчитать, сколько же пар скобок содержится в принятой строке. На этот вопрос каждый, кто реализует аппарат грамматик, отвечает по-своему, и в этом модуле он отвечен так. Разбор строки грамматикой разбивается на грамматические действия, как, например, выполнение процедуры разбора нетерминалом, выполнение процедуры разбора правилом, чтение нетерминала. Каждое такое грамматическое действие можно обрамить действиями над данными — например, можно модифицировать грамматику сбалансированных скобок, сделав в ней два нетерминала и три правила:

[17] Не ругайте за дубовость примера.

$S \to Q$, $Q \to (Q)Q$, $Q \to \lambda$, — начать процедуру “разобрать строку нетерминалом S” с действия “инициализировать промежуточную переменную нолём” и завершить процедуру “разобрать строку правилом $Q \to (Q)Q$” действием “увеличить промежуточную переменную на единицу”, и тогда по итогам успешного разбора строки в промежуточной переменной будет записано в точности то, что мы хотели посчитать.

Последнее замечание о грамматиках: часто разбор строки грамматикой происходит с учётом наличия так называемых “пробельных символов”. При разборе строки все пробельные символы игнорируются, то есть прочитываются без всякого эффекта. Например, пробельные символы языка `C++` — это пробел, табуляция и перевод строки.

12.1.2 Грамматики в модуле `grammar`

Модуль `grammar` предоставляет два способа работы с грамматиками: прямой и синтаксический.

Прямой способ подразумевает создание “чистой” грамматики (класс `Grammar`), в которой все компоненты неименованы, обращение к ним происходит через уникальные индексы (смещения в векторoподобных структурах).

Синтаксический способ создания грамматики (при помощи класса `syntax::Grammar`) более близок к математическому способу описания: некоторым компонентам грамматики можно присваивать уникальные имена (строки), и ссылаться на эти компоненты можно через эти имена. После создания синтаксической структуры описания грамматики достаточно использовать транслятор (класс `syntax::Interpreter`), чтобы автоматически получить чистую грамматику по её синтаксическому описанию. Все дополнительные действия, привносимые работой с синтаксисом и трансляцией, сосредоточены от начала создания объекта типа `syntax::Grammar` и до применения метода трансляции этого объекта в объект типа `Grammar`, после чего синтаксическое описание можно смело уничтожить.

Класс `Grammar` является производным от класса литералов (`regexp::Literal`).[18] Это означает, что грамматиками, как и регулярными выражениями, можно производить разбор строки, входного потока и возвратного потока (класса `stream::Stream`) с учётом и без учёта контекста разбора класса `regexp::Context` (регулярного контекста). Грамматика удовлетворяет всем соглашениям о разборе возвратного потока регулярными выражениями, кроме одного: если флаг активности данных поднят, но разбор завершился неуспехом, то не гарантирована неизменность данных. Это соглашение нарушается не просто так: грамматики могут иметь дело с довольно большими объёмами данных, а соблюдение соглашения о неизменности данных может привести к частому копированию данных, что будет работать ужасно неэффективно. Взамен пользователю предоставляется возможность “подчищения за собой”: действия над данными могут совершаться и при неуспешном завершении грамматических действий.

Класс `Grammar` дополнен методами разбора `gmatch`, аналогичными методам `match`, но вместо регулярного контекста принимающими на вход грамматический контекст (класс `Context`). Различие этих контекстов: регулярный контекст содержит хитрый указатель на произвольные данные (`type_abuse::DRef`), а грамматический вместо этого содержит совокупность данных (`dataset::Set`). Все соглашения о разборе с поправкой на одно нарушенное переносятся на методы `gmatch` с поправкой на то, что хотя флаг активности данных всё ещё один, но данных теперь много (и все они одновременно активируются и деактивируются этим флагом).

Разборы методами `match` и `gmatch` связываются между собой понятием “главных данных”. При запуске метода полного разбора `match_full`

- создаётся грамматический контекст со скопированными указателями на разобранную строку и флаг разбора строки;
- в новосозданном контексте заводится совокупность данных, определяемая грамматикой;
- главные данные в векторе совокупности перенаправляются на блок данных регулярного контекста разбора;
- флаг активности совокупности данных устанавливается равным флагу контекста для регулярных выражений (без перенаправления указателей);
- разбор делегируется методу полного разбора `gmatch_full`, работающему с получившимся грамматическим контекстом.

12.1.3 Чистая грамматика

Основное понятие, на котором основываются грамматики модуля `grammar`, — это чистое действие. Чистым действием определяется общий ход разбора. На данный момент в модуле реализованы следующие виды чистых действий:

1. Альтернатива (`alt`). В альтернативе содержится вектор индексов вершин грамматики (понятие вершины описано ниже). Выполнение этого действия — это последовательное выполнение его вершин до первого успеха. Альтернатива выполнена успешно, если какое-либо действие выполнилось успешно. Действие аналогично процедуре “разобрать поток нетерминалом”, в которой вместо правил выполняются произвольные вершины.
2. Правило (`rule`). В правиле содержится вектор индексов вершин грамматики. Выполнение этого действия — это последовательный запуск разбора вершинами из вектора. Каждая следующая вершина продолжает

[18] И не является “в чистом виде регулярным выражением”, так как это класс `Literal`, а не `Regexp`.

чтение с символа, на котором остановилась предыдущая. Правило выполнилось успешно, если все действия вектора выполнились успешно. Действие аналогично процедуре “разобрать поток правилом”, в которой вместо чтения терминалов и выполнения нетерминалов могут последовательно выполняться произвольные вершины.

3. Терминал (`re`). Выполнение этого действия — это запуск разбора пробельного выражения грамматики (о нём ниже) и затем — хранящегося в действии регулярного выраженя. Действие является обобщением чтения терминального символа в математической постановке: прочитать символ a — это выполнить действие `regexp::let(`a`)`.
4. Проброс (`next`). Выполнение этого действия — это выполнение вершины с индексом, содержащимся в действии.
5. Тривиальное действие. Это действие немедленно успешно завершает работу без изменения потока и контекста.

Последовательное выполнение чистых действий до первого успеха, диктуемое чистыми действиям, означает, что разбор совершается методом рекурсивного спуска с возвратом.

Грамматическое действие — это чистое действие, к которому по желанию может быть добавлено смещение связанных данных (в совокупности данных грамматического контекста). На данный момент это смещение игнорируется для всех действий. кроме терминала. Терминал без связанных данных выполняется в контексте с указателем на уникальные пустые данные и опущенным флагом данных в новом блоке. Терминал со связанными данными выполняется в контексте, указывающем на связанные данные и имеющем поднятый флаг данных в новом блоке. Указатели на разбираемую строку и флаг строки при разборе терминала направляются на таковые грамматического контекста.

Вершина грамматики состоит из четырех компонентов: грамматического действия и трёх функций над совокупностью данных (`dataset::Function`), а именно функции подготовки, функции завершения и функции очистки. Выполнение вершины происходит следующим образом. Выполняется функция подготовки (на совокупности данных текущего грамматического контекста). Затем выполняется действие. В случае успешного выполнения действия выполняется функция завершения, и выполнение вершины успешно завершается. В случае неуспешного выполнения действия выполняется функция очистки, и выполнение вершины неуспешно завершается.

Когда речь идёт о грамматиках в программировании, как правило появляется понятие пробельного символа — символа, который игнорируется при разборе и служит исключительно для форматирования текста и отделения конструкций друг от друга. В модуле `grammar` понятие пробельного символа обобщено до понятия пробельного регулярного выражения. Это выражение `r`, разбор которого запускается раз за разом до неуспеха непосредственно перед выполнением чистого действия нетерминала и не влияющее на успешность выполнения терминала. Данные контекста никак не изменяются пробельным выражением. Пробельные выражения учитываются в разобранной строке и возвращаются в поток при неуспехе. Например, в языке `C++` пробельными являются символы `' '`, `'
t'`, `'
n'`, и соответствующее пробельное выражение имеет вид `regexp::letset({' ', '
t', '
n'})`.

Грамматика (класса `Grammar`) состоит из проиндексированного набора вершин (каждой вершине присвоен уникальный номер), индекса начальной вершины, пробельного выражения, размера совокупности данных и — по желанию — индекса главных данных. При попытке запуска разбора `match` с требованием генерации данных и при этом невыставленным индексом главных данных немедленно будет возвращён неуспех.

12.1.4 Синтаксическое описание грамматики

Интерфейс работы с синтаксическими структурами расположен в подпространстве имён `syntax`. Создание чистой грамматики через этот интерфейс выглядит так:

1. Создать и наполнить синтаксическое описание грамматики — объект класса `syntax::Grammar`.
2. Создать транслятор — объект класса `syntax::Interpreter` — и получить чистую грамматику как результат выполнения метода `interpret` с аргументом — построенным синтаксическим описанием грамматики.

Синтаксическое описание грамматики наполняется синтаксическими описаниями компонентов чистой грамматики. В этих синтаксических описаниях запрещено использовать индексы вершин и индексы данных, и взамен этого разрешается:

1. Присваивать вершинам и данным имена: пространства имён вершин и данных различны, и внутри пространства каждый объект (вершина или данные, а точнее — будущие индекс вершины или индекс данных) связан с единственным именем, разным именам соответствуют разные объекты.
2. Явно задавать вложенные структуры, транслируемые в множества неименованных вершин (а не одну вершину).
3. Использовать имена вместо индексов в синтаксическом описании компонентов (вместо индексов).

Соглашения о трансляции имён данных в смещения в векторе данных описаны в модуле `dataset`. При этом не требуется создание синтаксического описания совокупности данных (`dataset::syntax::Set`) — такое описание автоматически создаётся при наполнении синтаксического описания грамматики из всех используемых имён

данных.

Если явно не указать имя начального действия при наполнении описания грамматики, то в результате трансляции будет получена чистая грамматика, начальная вершина которой тривиальна: содержит тривиальное действие и не содержит ничего более.

В описание грамматики можно добавить не только описание стандартного грамматического действия, но и описание нетерминала (`nonterminal`). Нетерминал задаётся вектором векторов описаний чистых действий. Внутренние векторы чистых действий считаются описаниями безымянных правил, внешний вектор объединяет эти правила описанием альтернативы, и с этой альтернативой связывается имя нетерминала.

Если имя использовалось в синтаксических описаниях компонентов грамматики, но с ним не была явна связана ни одна вершина, то с таким именем при трансляции связывается тривиальная вершина.

Связь имени с описанием вершины всегда можно устранить. Каждый метод, добавляющий вершину, устраняет связь с текущим описанием, после чего добавляет новое описание.

Если в описание грамматики не было добавлено пробельное выражение, то при трансляции подразумевается выражение `regexp::fail`, то есть отсутствие пробельных символов.

12.2 Интерфейс создания действий

Расположение

`grammar/action.hpp`

Собственно интерфейс

`Action trivial_action;`
Описание. Тривиальное действие.

`Action alt(const std::vector<size_t> & nodes);`
Описание. Вернуть альтернативу, содержащую копию вектора `nodes`.

`Action alt(std::vector<size_t> && nodes);`
Описание. Move-аналог предыдущей функции

`Action next(size_t node);`
Описание. Вернуть проброс, содержащий индекс `node`.

`Action re(const regexp::Regexp & re);`
Описание. Вернуть терминал, содержащий копию регулярного выражения `re`.

`Action re(regexp::Regexp && re);`
Описание. Move-аналог предыдущей функции

`Action rule(const std::vector<size_t> & nodes);`
Описание. Вернуть правило, содержащее копию вектора `nodes`.

`Action rule(std::vector<size_t> && nodes);`
Описание. Move-аналог предыдущей функции

12.3 `class Action;`

Расположение

`grammar/action.hpp`

Описание

Класс грамматического действия.

Конструкторы и присваивания по умолчанию, деструктор

Все в наличии, стандартная семантика с поправкой на действия по умолчанию над хитрыми указателями, содержащимися в регулярном выражении.
`Action()` инициализирует тривиальное чистое действие без индекса связанных данных.

Типы данных

`enum T {TRIVIAL, ALTERNATIVE, NEXT, REGEXP, RULE};`
Описание. `TRIVIAL` обозначает тривиальное действие.
`ALTERNATIVE` обозначает альтернативу.
`NEXT` обозначает проброс.
`REGEXP` обозначает терминал.
`RULE` обозначает правило.

Методы, изменяющие состояние объекта

`void set_alt(const std::vector<size_t> & nodes);`
Описание. Переустановить чистое действие в альтернативу, содержащую вектор `nodes`.

`void set_alt(std::vector<size_t> && nodes);`
Описание. Move-аналог предыдущего метода.

`void set_data(size_t data);`
Описание. Связать с действием индекс данных `data`.

`void set_next(size_t node);`
Описание. Переустановить чистое действие в проброс, содержащий индекс `next(node)`.

`void set_re(const regexp::Regexp & re);`
Описание. Переустановить чистое действие в терминал, содержащий регулярное выражение `re`.

`void set_re(regexp::Regexp && re);`
Описание. Move-аналог предыдущего метода.

`void set_rule(const std::vector<size_t> & nodes);`
Описание. Переустановить чистое действие в правило, содержащее вектор `nodes`.

`void set_rule(std::vector<size_t> && nodes);`
Описание. Move-аналог предыдущего метода.

`void unset();`
Описание. Переустановить чистое действие в тривиальное и отвязать индекс данных.

`void unset_control();`
Описание. Переустановить чистое действие в тривиальное.

`void unset_data();`
Описание. Отвязать индекс данных.

Методы доступа к типу

`T type() const;`
Описание. Вернуть тип установленного чистого действия.

Методы доступа к информации о наличии компонентов

`bool has_data() const;`
Описание. Вернуть `true` $\Leftrightarrow$ с действием связаны данные.

`bool has_next() const;`
Описание. Вернуть `true` $\Leftrightarrow$ в чистом действии содержится индекс следующей вершины.

`bool has_nodes() const;`
Описание. Вернуть `true` $\Leftrightarrow$ в чистом действии содержится вектор индексов вершин.

`bool has_re() const;`
Описание. Вернуть `true` $\Leftrightarrow$ в чистом действии содержится регулярное выражение.

Методы доступа к компонентам

`size_t & data();`
Описание. Вернуть индекс связанных данных.
Требования. Установлен индекс связанных данных.

`const size_t & data() const;`
Описание. Const-аналог предыдущего метода.

`size_t & next();`
Описание. Вернуть содержащийся индекс следующей вершины.
Требования. В действии содержится индекс следующей вершины.

`const size_t & next() const;`
Описание. Const-аналог предыдущего метода.

`std::vector<size_t> & nodes();`
Описание. Вернуть содержащийся вектор индексов вершин.
Требования. В действии содержится вектор индексов вершин.

`const std::vector<size_t> & nodes() const;`
Описание. Const-аналог предыдущего метода.

`regexp::Regexp & re();`

Описание. Вернуть содержащееся регулярное выражение.

Требования. В действии содержится регулярное выражение.

`const regexp::Regexp & re() const;`

Описание. Const-аналог предыдущего метода.

12.4 `class Context;`

Расположение

`grammar/context.hpp`

Описание

Класс контекста разбора.

Содержит два поля (подконтекста):

1. Активационный подконтекст разобранной строки `asc`.
2. Активационный подконтекст совокупности данных `adc`.

С учётом внутренней структуры полей, контекст содержит:

1. Хитрый указатель на разобранную строку.
2. Хитрый указатель на флаг активации строки.
3. Совокупность данных.
4. Хитрый указатель на флаг активации совокупности данных.

Конструкторы и присваивания по умолчанию, деструктор

Конструктор по умолчанию удалён.

Всё остальное в наличии, семантика стандартная с поправкой на действия по умолчанию над хитрыми указателями.

Другие конструкторы

`Context(size_t size);`

Описание. Инициализировать контекст указателем на новую пустую строку, совокупностью данных размера `size`, каждый элемент которой указывает на новый пустой блок, и опущенными флагами в новых уникальных блоках.

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

Описание. Инициализировать контекст совокупностью данных того же размера, что и в `adc`, и направить все указатели (строка, все данные, все флаги) на соответствующие блоки аргументов.

`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);`

Описание. Move-аналоги предыдущего конструктора.

Методы, изменяющие состояние объекта

`void detach();`

Описание. Открепить все указатели контекста (строка, данные, флаги).

`void detach_receive(const Context & sdc);`

Описание. Перенаправить указатели на строку и флаги контекста `*this` на клонов соответствующих блоков контекста `sdc` и выполнить метод `detach_receive` совокупности данных контекста `*this` с аргументом — совокупностью данных контекста `sdc`.

Требования. Наследуются от метода `detach_receive` совокупности данных: размер совокупности данных контекста `sdc` больше либо равен размеру совокупности данных контекста `*this`.

Методы, сохраняющие состояние объекта и изменяющие хранимые данные

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

Описание. Перезаписать блоки указателей на строку и флаги контекста `*this` соответствующими блоками контекста `sdc` и выполнить метод `receive` совокупности данных контекста `*this` с аргументом — совокупностью данных контекста `sdc`.

Требования. Наследуются от метода `receive` совокупности данных: размер совокупности данных контекста `sdc` больше либо равен размеру совокупности данных контекста `*this`.

Методы, сохраняющие состояние объекта и хранимые данные

`Context clone() const;`
Описание. Вернуть контекст с таким же размером совокупности данных, что и в `*this`, все указатели которого направлены на клонов блоков соответвующих указателей контекста `*this`.

`void send(const Context & sdc) const;`
Описание. Перезаписать блоки указателей на строку и флаги контекста `sdc` соответствующими блоками контекста `*this` и выполнить метод `send` совокупности данных контекста `*this` с аргументом — совокупностью данных контекста `sdc`.
Требования. Наследуются от метода `send` совокупности данных: размер совокупности данных контекста `sdc` больше либо равен размеру совокупности данных контекста `*this`.

Методы доступа

`type_abuse::DRef & data(size_t i);`
Описание. Вернуть указатель на данные с индексом `i` в совокупности.
Требования. Размер совокупности данных больше либо равен `i+1`.

`const type_abuse::DRef & data(size_t i) const;`
Описание. Const-аналог предыдущего метода.

`size_t data_size() const;`
Описание. Вернуть размер совокупности данных.

```
template<typename Value>
Value & data_value(size_t i) const;
```
Описание. Вернуть данные с индексом `i` в совокупности, трактуемые как данные типа `Value`.
Требования. Размер совокупности данных больше либо равен `i+1`.
Блок данных с индексом `i` в совокупности непуст и хранит данные типа `Value`.

`bool is_active() const;`
Описание. Вернуть `true` $\Leftrightarrow$ хотя бы один из флагов поднят.

`bool is_data_empty(size_t i) const;`
Описание. Вернуть `true` $\Leftrightarrow$ блок указателя на данные с индексом `i` в совокупности пуст.

`bool is_data_nonempty(size_t i) const;`
Описание. Вернуть `true` $\Leftrightarrow$ блок указателя на данные с индексом `i` в совокупности непуст.

`bool is_inactive() const;`
Описание. Вернуть `true` $\Leftrightarrow$ оба флага опущены.

`std::string & string() const;`
Описание. Вернуть разобранную строку.

Поля

`regexp::context::AString asc;`
Описание. Подконтекст, содержащий хитрый указатель на разобранную строку и активатор — хитрый указатель на флаг активности строки.

`context::ADataSet adc;`
Описание. Подконтекст, содержащий совокупность данных и активатор — хитрый указатель на флаг активности совокупности данных.

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

Расположение

`grammar/grammar.hpp`

Описание

Класс грамматики (или, более развёрнуто, — чистой грамматики).
Методы этого класса не позволяют удобно устанавливать функции над совокупностью данных в вершины. Сделать это можно двумя способами. Первый: подготовить вершину с установленными функциями и добавить её в грамматику специальным методом. Второй: специальным методом получить ссылку на добавленную ранее вершину, и добавить желаемые функции, используя методы этой вершины.
При добавлении чистого действия в грамматику добавляется вершина, содержащая только это действие и ничего более — ни индекса связанных данных, ни функций над совокупностью данных. При добавлении новой вершины всегда возвращается индекс, отличающийся от всех возвращавшихся ранее.
Кроме явно описанных далее методов также наследуются методы разбора класса `regexp::Literal`, кроме метода полного разбора. Метод полного разбора перегружен, как сказано в общем описании аппарата грамматик.

Конструкторы и присваивания по умолчанию, деструктор

Все в наличии, стандартная семантика с поправкой на действия по умолчанию над хитрыми указателями, содержащимися в регулярных выражениях.
`Grammar()` инициализирует грамматику с одной тривиальной начальной вершиной, пробельным выражением `regexp::fail` (то есть отсутствием пробельных конструкций), совокупностью данных нолевого размера и отсутствующим смещением главных данных.

Методы, добавляющие новые действия

`size_t push_alt(const std::vector<size_t> & nodes);`
Описание. Добавить в грамматику альтернативу, содержащую вектор вершин `nodes`.
Вернуть индекс добавленной вершины.

`size_t push_alt(std::vector<size_t> && nodes);`
Описание. Move-аналог предыдущего метода.

`size_t push_next(size_t node);`
Описание. Добавить в грамматику проброс, содержащий вершину `node`.
Вернуть индекс добавленной вершины.

`size_t push_node();`
Описание. Добавить в грамматику тривиальное действие.
Вернуть индекс добавленной вершины.

`size_t push_node(const Node & node);`
Описание. Добавить в грамматику копию вершины `node`.
Вернуть индекс добавленной вершины.

`size_t push_node(Node && node);`
Описание. Move-аналог предыдущего метода.

`size_t push_re(const regexp::Regexp & r);`
Описание. Добавить в грамматику терминал, содержащий выражение `r`.
Вернуть индекс добавленной вершины.

`size_t push_re(regexp::Regexp && re);`
Описание. Move-аналог предыдущего метода.

`size_t push_rule(const std::vector<size_t> & nodes);`
Описание. Добавить в грамматику правило, содержащее вектор `rule(nodes)`.
Вернуть индекс добавленной вершины.

`size_t push_rule(std::vector<size_t> && nodes);`
Описание. Move-аналог предыдущего метода.

Методы, изменяющие состояние объекта, кроме добавления новых действий

`void set_data_size(size_t size);`
Описание. Установить размер совокупности данных контекста в `size`.

`void set_main_data_index(size_t i);`
Описание. Установить индекс главных данных в `i`.

`void set_main_index(size_t i);`
Описание. Установить индекс начальной вершины грамматики в `i`.

`void set_nodes(const std::vector<Node> & nodes);`
Описание. Перезаписать вектор вершин грамматики вектором `nodes`.

`void set_nodes(std::vector<Node> && nodes);`

Описание. Move-аналог предыдущего метода.

`void set_skip(const regexp::Regexp & re);`
Описание. Установить пробельное выражение в `re`.

`void set_skip(regexp::Regexp && re);`
Описание. Move-аналог предыдущего метода.

`void unset_main_data_index();`
Описание. Удалить информацию об индексе главных данных.

Методы разбора

`bool gmatch(stream::Stream & s, Context & gcontext = default_context) const;`
Описание. Вызвать метод `gmatch_full` с теми же аргументами.
Вернуть результат выполнения этого метода.

`bool gmatch(std::istream & s, Context & gcontext = default_context) const;`
Описание. Вызвать метод `gmatch` с возвратным потоком `stream::Stream(s)` и тем же контекстом.
Вернуть результат выполнения этого метода.

`bool gmatch(const std::string & s, Context & gcontext = default_context) const;`
Описание. Вызвать метод `gmatch` со стандартным потоком ввода — базой потока `std::stringstream(s)` — и тем же контекстом.
Вернуть результат выполнения этого метода.

`bool gmatch_full(stream::Stream & s, Context & gcontext) const;`
Описание. Запустить разбор грамматики методом рекурсивного спуска с возвратом с начальной вершиной с установленным индексом.
Требования. Вершина с индексом, равным установленному индексу начальной вершины, существует в грамматике.
Вершины со всеми индексами, используемыми в добавленных в грамматику вершинах, существуют в грамматике.
Если установлен флаг активности совокупности данных контекста `context`, то для каждого индекса данных `i`, используемого в грамматике, размер совокупности данных контекста `context` больше либо равен `i+1`.

`bool match_full(stream::Stream & s, regexp::Context & context) const override;`
Описание. Если флаг активности данных контекста `context` поднят и индекс главных данных не установлен, то вернуть `false`, не затрагивая поток и контекст.
В противном случае вызвать `gmatch_full` с тем же потоком и с контекстом, созданным по `context` так, как представлено в общем описании, и вернуть результат вызова.
Требования. Вершина с индексом, равным установленному индексу начальной вершины, существует в грамматике.
Вершины со всеми индексами, используемыми в добавленных в грамматику вершинах, существуют в грамматике.
Если установлен индекс главных данных `i`, то размер совокупности данных грамматики больше либо равен `i+1`.
Если установлен флаг активности данных контекста `context`, то для каждого индекса данных `i`, используемого в функциях работы с данными, добавленных в грамматику, размер совокупности данных грамматики больше либо равен `i+1`.

Методы доступа

`size_t data_size() const;`
Описание. Вернуть установленный размер совокупности данных.

`bool has_main_data() const;`
Описание. Вернуть `true` ⇔ установлен индекс главных данных.

`size_t main_data_index() const;`
Описание. Вернуть индекс главных данных.
Требования. Индекс главных данных установлено.

`size_t main_index() const;`
Описание. Вернуть индекс начальной вершины.

`Node & main_node();`
Описание. Вернуть начальную вершину.
Требования. Вершина с индексом начальной вершины существует в грамматике.

`const Node & main_node() const;`

Описание. Const-аналог предыдущего метода.

`Node & node(size_t i);`
Описание. Вернуть вершину с индексом `i`.
Требования. Вершина с индексом `i` существует в грамматике.

`const Node & node(size_t i) const;`
Описание. Const-аналог предыдущего метода.

`regexp::Regexp skip() const;`
Описание. Вернуть пробельное регулярное выражение.

12.6 `class Node;`

Расположение

`grammar/node.hpp`

Описание

Класс вершины грамматики.
Каждая из четырёх составляющих вершины (действие и три функции над совокупностью данных) может отсутствовать. Если отсутствует функция над совокупностью данных, то в соответствующий момент совокупность данных не изменяется. Отсутствующее действие при разборе трактуется как тривиальное действие, не связанное с данными.

Конструкторы и присваивания по умолчанию, деструктор

Все в наличии, стандартная семантика с поправкой на действия по умолчанию над хитрыми указателями, содержащимися в регулярном выражении внутри действия.
`Node()` инициализирует вершину, в которой отсутствуют все компоненты.

Другие конструкторы

`Node(const Action & act);`
Описание. Инициализирует действие вершины копией действия `act`.
Все функции отсутствуют.

`Node(Action && act);`
Описание. Move-аналог предыдущего конструктора.

Методы установки составляющих вершины

`void set_action();`
Описание. Установить тривиальное действие.

`void set_action(const Action & act);`
Описание. Установить копию действия `act`.

`void set_action(Action && act);`
Описание. Move-аналог предыдущего метода.

`void set_post_fail();`
Описание. Установить тривиальную функцию очистки.

`void set_post_fail(const dataset::Function & fun);`
Описание. Установить копию `fun` как функцию очистки.

`void set_post_fail(dataset::Function && fun);`
Описание. Move-аналог предыдущего метода.

`void set_post_success();`
Описание. Установить тривиальную функцию завершения.

`void set_post_success(const dataset::Function & fun);`
Описание. Установить копию `fun` как функцию завершения.

`void set_post_success(dataset::Function && fun);`
Описание. Move-аналог предыдущего метода.

`void set_pre();`
Описание. Установить тривиальную функцию подготовки.

`void set_pre(const dataset::Function & fun);`
Описание. Установить копию `fun` как функцию подготовки.

`void set_pre(dataset::Function && fun);`
Описание. Move-аналог предыдущего метода.

Методы удаления составляющих вершины

`void unset();`
Описание. Удалить все составляющие вершины.

`void unset_action();`
Описание. Удалить действие.

`void unset_post_fail();`
Описание. Удалить функцию очистки.

`void unset_post_success();`
Описание. Удалить функцию завершения.

`void unset_pre();`
Описание. Удалить функцию подготовки.

Методы проверки наличия составляющих вершины

`bool has_act() const;`
Описание. Вернуть `true` ⇔ действие установлено.

`bool has_post_fail() const;`
Описание. Вернуть `true` ⇔ функция очистки установлена.

`bool has_post_success() const;`
Описание. Вернуть `true` ⇔ функция завершения установлена.

`bool has_pre() const;`
Описание. Вернуть `true` ⇔ функция подготовки установлена.

Методы доступа к составляющим вершины

`Action & act();`
Описание. Вернуть установленное действие.
Требования. Действие установлено.

`const Action & act() const;`
Описание. Const-аналог предыдущего метода.

`dataset::Function & post_fail();`
Описание. Вернуть установленную функцию очистки.
Требования. Функция очистки установлена.

`const dataset::Function & post_fail() const;`
Описание. Const-аналог предыдущего метода.

`dataset::Function & post_success();`
Описание. Вернуть установленную функцию завершения.
Требования. Функция завершения установлена.

`const dataset::Function & post_success() const;`
Описание. Const-аналог предыдущего метода.

`dataset::Function & pre();`
Описание. Вернуть установленную функцию подготовки.
Требования. Функция подготовки установлена.

`const dataset::Function & pre() const;`
Описание. Const-аналог предыдущего метода.

12.7 `class ADataSet;`

Расположение

`grammar/context/adataset.hpp`

Пространство имён описания

`pm::grammar::context`

Описание

Класс подконтекста, используемого в качестве поля грамматического контекста разбора.
Содержит активатор (хитрый указатель на булево значение — флаг активности) и совокупность данных.
Совокупность данных активна, если флаг активатора поднят.
Совокупность данных неактивна, если флаг активатора опущен.

Конструкторы и присваивания по умолчанию, деструктор

Конструктор без аргументов удалён.
Остальные конструкторы, присваивания и деструктор в наличии, семантика стандартное с поправкой на действия по умолчанию над хитрыми указателями.

Другие конструкторы

`ADataSet(bool act, size_t size);`
Описание. Направить активатор на обёртку над `act`.
Инициализировать совокупность данных размера `size`, каждый указатель которой направлен на новый уникальный пустой блок.

`ADataSet(const regexp::context::Activator & act, const dataset::Set & set);`
Описание. Завести совокупность данных того же размера, что и `set`.
Направить все хитрые указатели `*this` на блоки соответствующих указателей аргументов.

`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);`
Описание. Move-аналоги предыдущего конструктора.

Другие операторы

`type_abuse::DRef & operator [](size_t i);`
Описание. Вернуть данные по смещению `i`.
Требования. Размер совокупности данных больше или равен `i+1`.

`const type_abuse::DRef & operator [](size_t i) const;`
Описание. Const-аналог предыдущего метода.

Методы, изменяющие состояние объекта

`void detach();`
Описание. Открепить все хитрые указатели.

`void detach_receive(const ADataSet & adc);`
Описание. Перенаправить активатор на клона блока активатора подконтекста `adc`.
Выполнить метод `detach_receive` совокупности данных подконтекста `*this` с аргументом — совокупностью данных подконтекста `adc`.
Требования. Наследуются от типа совокупности данных: размер совокупности данных подконтекста `adc` больше или равен размеру совокупности данных подконтекста `*this`.

`void detach_unset_activate();`
Описание. Перенаправить активатор на обёртку над `true`.
Перенаправить все указатели совокупности данных на новые пустые блоки.

`void detach_unset_deactivate();`
Описание. Перенаправить активатор на обёртку над `true`.
Перенаправить все указатели совокупности данных на новые пустые блоки.

Методы, сохраняющие состояние объекта и изменяющие хранимые данные

`void receive(const ADataSet & adc) const;`
Описание. Перезаписать блок активатора подконтекста `*this` блоком активатора подконтекста `adc`.
Выполнить метод `receive` совокупности данных подконтекста `*this` с аргументом — совокупностью данных подконтекста `adc`.
Требования. Наследуются от типа совокупности данных: размер совокупности данных подконтекста `adc` больше или равен размеру совокупности данных подконтекста `*this`.

`void unset_activate() const;`
Описание. Поднять флаг активности.
Опустошить все указатели на данные.

`void unset_deactivate() const;`
Описание. Опустить флаг активности.
Опустошить все указатели на данные.

Методы, сохраняющие состояние объекта и хранимые данные

`ADataSet clone() const;`
Описание. Вернуть подконтекст с таким же размером совокупности данных, как и в `*this`, все указатели которого направлены на клонов блоков соответствующих указателей подконтекста `*this`.

`void send(const ADataSet & adc) const;`
Описание. Перезаписать блок активатора подконтекста `adc` блоком активатора подконтекста `*this`. Выполнить метод `send` совокупности данных подконтекста `*this` с аргументом — совокупностью данных подконтекста `adc`.
Требования. Наследуются от типа совокупности данных: размер совокупности данных подконтекста `adc` больше или равен размеру совокупности данных подконтекста `*this`.

Методы доступа

`type_abuse::DRef & at(size_t i);`
Описание. Вернуть указатель на данные по смещению `i` в совокупности.
Требования. Размер совокупности данных больше или равен `i+1`.

`const type_abuse::DRef & at(size_t i) const;`
Описание. Const-аналог предыдущего метода.

`bool is_active() const;`
Описание. Вернуть `true` ⇔ флаг активности поднят.

`bool is_empty(size_t i) const;`
Описание. Вернуть `true` ⇔ блок указателя на данные по смещению `i` в совокупности пуст.
Требования. Размер совокупности данных больше или равен `i+1`.

`bool is_inactive() const;`
Описание. Вернуть `true` ⇔ флаг активности опущен.

`bool is_nonempty(size_t i) const;`
Описание. Вернуть `true` ⇔ блок указателя на данные по смещению `i` в совокупности непуст.
Требования. Размер совокупности данных больше или равен `i+1`.

`size_t size() const;`
Описание. Вернуть размер совокупности данных.

```
template<typename Value>
Value & value(size_t i) const;
```
Описание. Вернуть данные блока указателя по смещению `i` в совокупности данных, трактуемые как данные типа `Value`.
Требования. Размер совокупности данных больше или равен `i+1`.
Блок указателя по смещению `i` в совокупности данных непуст и хранит значение типа `Value`.

Поля

`regexp::context::Activator ac;`
Описание. Активатор (хитрый указатель на значение типа `bool`) совокупности данных.

`dataset::Set dc;`
Описание. Совокупность данных.

12.8 Интерфейс создания синтаксических описаний чистых действий

Расположение

`grammar/syntax/action.hpp`

Пространство имён описания

`pm::grammar::syntax`

Общее описание

Интерфейс позволяет получать объекты типа `Action` — синтаксические описания чистых действий, используемые в качестве основных строительных блоков синтаксического описания грамматики. В описании интерфейса говорится, в какое действие транслируется получаемое синтаксическое описание чистого действия. Чтобы сказать это покороче, используются такие сокращения: `<vector>` — вектор, содержащий индексы вершин, в которые транслируются элементы вектора `vector`; `<string>` — индекс вершины, получаемой в результате трансляции имени `string`.

Собственно интерфейс

`Action trivial_action;`
Соответствие. Тривиальное действие.

`Action alt(const std::vector<Action> & actions);`
Соответствие. `grammar::alt(<actions>)`.

`Action alt(std::vector<Action> && actions);`
Описание. Move-аналог предыдущей функции.

`Action name(const std::string & nm);`
Соответствие. `grammar::next(<nm>)`.

`Action name(std::string && nm);`
Описание. Move-аналог предыдущей функции.

`Action re(const regexp::Regexp & r);`
Соответствие. `grammar::re(r)`.

`Action re(regexp::Regexp && r);`
Описание. Move-аналог предыдущей функции.

`Action rule(const std::vector<Action> & actions);`
Соответствие. `grammar::rule(<actions>)`.

`Action rule(std::vector<Action> && actions);`
Описание. Move-аналог предыдущей функции.

12.9 `class syntax::Action;`

Расположение

`grammar/syntax/action.hpp`

Пространство имён описания

`pm::grammar::syntax`

Описание

Класс синтаксического описания чистых действий.
Пользователям не рекомендуется использование внутреннего интерфейса этого класса, так как есть более удобный интерфейс создания синтаксических описаний действий. Но если вдруг кому-то зачем-то понадобится описание этого класса, то вот оно.

Конструкторы и присваивания по умолчанию, деструктор

Все в наличии, стандартная семантика с поправкой на действия по умолчанию над хитрыми указателями, содержащимися в регулярных выражениях.
`Action()` инициализирует описание действия `trivial_action`.

Типы

`enum T TRIVIAL, ALTERNATIVE, NAME, REGEXP, RULE;`
Описание. `TRIVIAL` обозначает описание тривиального действия.
`ALTERNATIVE` обозначает описание альтернативы.
`NAME` обозначает описание проброса.
`REGEXP` обозначает описание терминала.
`RULE` обозначает описание правила.

Методы, изменяющие состояние объекта

`void set_alt(const std::vector<Action> & actions);`
Описание. Установить описываемое действие в `alt(actions)`.

`void set_alt(std::vector<Action> && actions);`
Описание. Move-аналог предыдущего метода.

`void set_name(const std::string & nm);`
Описание. Установить описываемое действие в `name(nm)`.

`void set_name(std::string && nm);`
Описание. Move-аналог предыдущего метода.

`void set_re(const regexp::Regexp & r);`
Описание. Установить описываемое действие в `re(r)`.

`void set_re(regexp::Regexp && r);`
Описание. Move-аналог предыдущего метода.

`void set_rule(const std::vector<Action> & actions);`
Описание. Установить описываемое действие в `rule(actions)`.

`void set_rule(std::vector<Action> && actions);`
Описание. Move-аналог предыдущего метода.

`void unset();`
Описание. Установить описываемое действие в тривиальное.

Методы доступа к типу

`T type() const;`
Описание. Вернуть тип описываемого действия.

Методы доступа к информацию о наличии компонентов

`bool has_actions() const;`
Описание. Вернуть `true` $\Leftrightarrow$ в описываемом действии содержится вектор описаний действий.

`bool has_name() const;`
Описание. Вернуть `true` $\Leftrightarrow$ в описываемом действии содержится имя.

`bool has_re() const;`
Описание. Вернуть `true` $\Leftrightarrow$ в описываемом действии содержится регулярное выражение.

Методы доступа к компонентам

`std::vector<Action> & actions();`
Описание. Вернуть содержащийся вектор описаний действий.
Требования. В описываемом действии содержится вектор описаний действий.

`const std::vector<Action> & actions() const;`
Описание. Const-аналог предыдущего метода.

`std::string & name();`
Описание. Вернуть содержащееся имя.
Требования. В описываемом действии содержится имя.

`const std::string & name() const;`
Описание. Const-аналог предыдущего метода.

`regexp::Regexp & re();`
Описание. Вернуть содержащееся регулярное выражение.
Требования. В описываемом действии содержится регулярное выражение.

`const regexp::Regexp & re() const;`
Описание. Const-аналог предыдущего метода.

12.10 `class syntax::Grammar;`

Расположение

`grammar/syntax/grammar.hpp`

Пространство имён описания

`pm::grammar::syntax`

Описание

Класс синтаксического описания грамматики.
Объекты этого класса автоматически транслируются в (чистые) грамматики.
Класс содержит полное синтаксическое описание всех составных частей грамматики: множество описаний вершин (некоторым из которых соответствуют имена), имя главной вершины, имя главных данных, пробельное регулярное выражение.
Методы добавления описания вершины автоматически связывают добавляемое описание с именем (строкой). Как только имя вершины появляется в описании грамматики (в методе добавления вершины, или в синтаксическом описании добавляемой вершины), с ним немедленно связывается полное описание вершины. Описание вершины содержит описания всех компонентов вершины: чистого действия, индекса связанных данных и функций подготовки, завершения и очистки. Каждая часть описания считается неустановленной, если не была установлена явно соответствующими методами. Неустановленное описание чистого действия при трансляции означает тривиальное чистое действие, неустановленное описание связанных данных — отсутствие индекса связанных данных, неустановленное описание фунции над совокупностью данных — тривиальную функцию. Если при добавлении компонента вершины (или вершины целиком) вершина с заданным именем существовала, то соответствующий компонент (вся вершина) переустанавливается.
При наполнении описания грамматики можно использовать как имена, с которыми уже связаны компоненты описания вершины, так и имена, с которыми эти компоненты будут связываться позднее. При этом (согласно написанному выше) в момент использования имени, с которым ещё ничего не связано, с этим именем связывается вершина, все компоненты которой не установлены.

Конструкторы и присваивания по умолчанию, деструктор

Все в наличии, имеют стандартную семантику с поправкой на действия по умолчанию над хитрыми указателями, содержащимися в регулярных выражениях.
`Grammar()` инициализирует грамматику, в которой не установлены никакие компоненты.

Методы добавления действий

`void set_action(const std::string & name, const Action & action);`
Описание. Добавить описание чистого действия `action`, связав его с именем `name`.

`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);`
Описание. Move-аналоги предыдущего метода.

`void set_alt(const std::string & name, const std::vector<Action> & actions);`
Описание. Добавить описание чистого действия `alt(actions)`, связав его с именем `name`.

`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);`
Описание. Move-аналоги предыдущего метода.

`void set_next(const std::string & name, const std::string & next_name);`
Описание. Добавить описание чистого действия `next(next_name)`, связав его с именем `name`.

`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);`
Описание. Move-аналоги предыдущего метода.

`void set_nonterminal(const std::string & name, const std::vector<std::vector<Action» & action_matrix);`
Описание. Добавить описание чистого действия `alt({rule(action_matrix[0]), rule(action_matrix[1]), ..., rule(action_matrix[k-1])})`, где `k` — размер внешнего вектора `action_matrix`, связав это описание с именем `name`.

`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);`

Описание. Move-аналоги предыдущего метода.

`void set_re(const std::string & name, const regexp::Regexp & r);`
Описание. Добавить описание чистого действия `re(r)`, связав его с именем `name`.

`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);`
Описание. Move-аналоги предыдущего метода.

`void set_rule(const std::string & name, const std::vector<Action> & actions);`
Описание. Добавить описание чистого действия `rule(actions)`, связав его с именем `name`.

`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);`
Описание. Move-аналоги предыдущего метода.

Методы установки данных и функций вершины

`void set_data(const std::string & name, const std::string & data);`
Описание. Связать данные с именем `data` с вершиной с именем `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);`
Описание. Move-аналоги предыдущего метода.

`void set_post_fail(const std::string & name, const dataset::syntax::Function & fun);`
Описание. Установить функцию очистки с описанием `fun` в вершину с именем `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);`
Описание. Move-аналоги предыдущего метода.

`void set_post_success(const std::string & name, const dataset::syntax::Function & fun);`
Описание. Установить функцию завершения `fun` в вершину с именем `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);`
Описание. Move-аналоги предыдущего метода.

`void set_pre(const std::string & name, const dataset::syntax::Function & fun);`
Описание. Установить функцию подготовки `fun` в вершину с именем `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);`
Описание. Move-аналоги предыдущего метода.

Методы установки глобальных компонентов

`void set_main_data(const std::string & name);`
Описание. Установить имя главных данных `name`.

`void set_main_data(std::string && name);`
Описание. Move-аналог предыдущего метода.

`void set_main_name(const std::string & name);`
Описание. Установить имя начальной вершины `name`.
Если вершины с таким именем нет в описании грамматики, то связать с этим именем пустую вершину.

`void set_main_name(std::string && name);`
Описание. Move-аналог предыдущего метода.

`void set_skip(const regexp::Regexp & re);`
Описание. Установить пробельное выражение `re`.

`void set_skip(regexp::Regexp && re);`
Описание. Move-аналог предыдущего метода.

Методы удаления компонентов

`void unset(const std::string & name);`
Описание. Удалить имя вершины `name` и связанное с ним описание вершины, если таковые есть в описании грамматики.

`void unset_action(const std::string & name);`
Описание. Если с именем `name` не связано описание вершины, то ничего не делать.
Иначе удалить действие вершины с именем `name`, если оно установлено.

`void unset_data(const std::string & name);`
Описание. Если с именем `name` не связано описание вершины, то ничего не делать.
Иначе удалить имя данных, связанных с вершиной с именем `name`, если оно установлено.

`void unset_main_data();`
Описание. Удалить имя главных данных, если оно установлено.

`void unset_main_name();`
Описание. Удалить имя начальной вершины, если оно установлено.
При этом само имя и связанное с ним описание вершины остаются в описании грамматики.

`void unset_post_fail(const std::string & name);`
Описание. Если с именем `name` не связано описание вершины, то ничего не делать.
Иначе удалить функцию очистки вершины `name`, если она была установлена.

`void unset_post_success(const std::string & name);`
Описание. Если с именем `name` не связано описание вершины, то ничего не делать.
Иначе удалить функцию завершения вершины `name`, если она была установлена.

`void unset_pre(const std::string & name);`
Описание. Если с именем `name` не связано описание вершины, то ничего не делать.
Иначе удалить функцию подготовки вершины `name`, если она была установлена.

`void unset_skip();`
Описание. Удалить пробельное регулярное выражение.

Методы проверки наличия компонентов

`bool has_main_data() const;`
Описание. Вернуть `true` $\Leftrightarrow$ имя главных данных установлено.

`bool has_main_node() const;`
Описание. Вернуть `true` $\Leftrightarrow$ имя начальной вершины установлено.

`bool has_node(const std::string & name) const;`
Описание. Вернуть `true` $\Leftrightarrow$ имя `name` связано с каким-либо описанием вершины.

`bool has_skip() const;`
Описание. Вернуть `true` $\Leftrightarrow$ пробельное выражение установлено.

Методы доступа к компонентам

`std::string & main_data();`
Описание. Вернуть имя главных данных.
Требования. Имя главных данных установлено.

`const std::string & main_data() const;`
Описание. Const-аналог предыдущего метода.

`const std::string & main_name() const;`
Описание. Вернуть имя начальной вершины.
Требования. Имя начальной вершины установлено.

`Node & main_node();`
Описание. Вернуть описание начальной вершины.
Требования. Имя начальной вершины установлено.

`const Node & main_node() const;`
Описание. Const-аналог предыдущего метода.

`Node & node(const std::string & name);`
Описание. Вернуть описание вершины, связанное с именем `name`.
Требования. Имя `name` связано с каким-либо описанием вершины.

`const Node & node(const std::string & name) const;`
Описание. Const-аналог предыдущего метода.

`const utils::Numeration<std::string> & node_names() const;`

Описание. Вернуть нумерацию, область определения которой — все имена описаний вершин, содержащиеся в описании грамматики.

`regexp::Regexp & skip();`
Описание. Вернуть пробельное выражение.
Требования. Пробельное выражение установлено.

`const regexp::Regexp & skip() const;`
Описание. Const-аналог предыдущего метода.

12.11 `class syntax::Interpreter;`

Расположение

`grammar/syntax/interpreter.hpp`

Пространство имён описания

`pm::grammar::syntax`

Описание

Класс трансляторов грамматик.
Процесс трансляции представлен в общем описании модуля и в описании класса `Grammar`.
Для трансляции грамматики достаточно создать объект транслятора и вызвать метод `interpret`, подав в него итоговое синтаксическое описание грамматики. Для многократной трансляции можно без опаски многократно вызывать этот метод с разными описаниями.
Второй метод — `reset` — на данный момент является избыточным, так как является составной частью метода `interpret`.

Конструкторы и присваивания по умолчанию, деструктор

Все в наличии, стандартная семантика.

Методы

`grammar::Grammar interpret(const Grammar & sgr);`
Описание. Возвращает чистую грамматику, являющуюся результатом трансляции синтаксического описания грамматики `sgr`.

`void reset();`
Описание. Бесполезный метод, сбрасывающий внутренние структуры данных транслятора в начальное состояние. Является составной частью метода `interpret`.

12.12 `struct syntax::Node;`

Расположение

`grammar/syntax/node.hpp`

Пространство имён описания

`pm::grammar::syntax`

Описание

Класс синтаксического описания вершины грамматики.
Пользователям не рекомендуется использование этого класса, так как для обычных нужд, возникающих при работе с синтаксическим описанием грамматики, хватает интерфейса класса `Grammar`.
Но если вдруг кому-то зачем-то понадобится именно этот класс: на самом деле это просто структура, содержащая опционалы, хранящие описание всех составляющих частей вершины (чистого действия, связанных данных и функций подготовки, завершения и очистки).

Поля

`utils::Optional<Action> action`;

Описание. Непустой опционал содержит синтаксическое описание чистого действия.
Пустой опционал означает описание тривиального действия.

`utils::Optional<std::string> data`;

Описание. Непустой опционал содержит имя данных совокупности грамматического контекста разбора, связанных с действием.
Пустой опционал означает отсутствие связанных данных.

`utils::Optional<dataset::syntax::Function> post_fail`;

Описание. Непустой опционал содержит синтаксическое описание функции очистки.
Пустой опционал означает тривиальную функцию.

`utils::Optional<dataset::syntax::Function> post_success`;

Описание. Непустой опционал содержит синтаксическое описание функции завершения.
Пустой опционал означает тривиальную функцию.

`utils::Optional<dataset::syntax::Function> pre`;

Описание. Непустой опционал содержит синтаксическое описание функции подготовки.
Пустой опционал означает тривиальную функцию.