Содержание

Введение

Этот документ предназначен для того, чтобы помочь людям, которым пришлось столкнуться с библиотекой Lspl разобраться с тем, как с ним работать.

Этот документа находится в стадии разработки, так что приветствуются любые комментарии (замечания, предложения и т.п.) — их можно высылать на адрес **alexey.noskov@gmail.com**.

Сборка из исходников

Здесь описывается установка из репозитория для Ubuntu.

Для начала необходимо установить зависимости проекта:

- boost — библиотека общено назначения;
- pcrecpp — библиотека поддержки регулярных выражений;
- cmake — используется для сборки проекта.

В Ubuntu для установки достаточно вызвать:

```
sudo apt-get install libpcre3-dev libboost-dev cmake
```

Если же планируется собирать модули для Java, то необходимо дополнительно установить JRE и ant. Для этого можно использовать следующие команды:

```
sudo apt-get install sun-java6-jre ant
```

Для доступа к исходному коду в репозитории необходимо установить git:

```
sudo apt-get install git
```

После того, как все установлено можно скачивать исходный код с сервера:

```
git clone git://gitorious.org/lspl/lspl.git
```

После выполнения команды в текущей директории появится директория lspl, содержащая исходный код основных модулей.

Теперь необходимо собрать код. Самый простой способ это сделать — использовать Makefile, находящийся в директории lspl. В нем определены следующие основные цели:

- **all** (цель по умолчанию) — собрать все модули;
- **core** — собрать ядро;
- **tools** — собрать утилиты;
- **java** — собрать интерфейсную библиотеку для Java;
- **gui** — собрать графическую оболочку.

Например, если вы планируете использовать ядро и утилиты, вызовите:

```
make tools
```

Поскольку утилиты зависят от ядра, то ядро будет собрано автоматически.

Создание простейшего приложения

Здесь я опишу создание простейшего приложения, которое осуществляет поиск некоторого шаблона в тексте. Пусть, для начала имеется файл simple.cpp следующего содержания:

```
#include <iostream>


int main() {
  std::cout << "Application started" << std::endl;


  std::cout << "Application finished" << std::endl;


  return 0;
}
```

Также, пусть для сборки используется следующий Makefile:

```
simple: simple.cpp
	g++ simple.cpp -o simple
```

Подключение Lspl

Для начала, необходимо подключить библиотеку и проверить, что все корректно собирается. Для этого допишем в начало **simple.cpp** следующий код (пока его смысл не так важен — главное, что он использует Lspl):

```
#include <lspl/Namespace.h>
```

А в тело main:

```
lspl::NamespaceRef ns = new lspl::Namespace();
```

Теперь, чтобы приложение корректно собиралось необходимо прописать пути к заголовочным файлам Lspl и подключить библиотеку. Пусть Lspl был собран в директории <LSPL>. Тогда, для указания путей к заголовочным файлам, в вызов g++ необходимо добавить следующие параметры:

```
-I <LSPL>/core/src/main
```

Для подключения библиотеки необходимо добавить параметры:

```
-L <LSPL>/core/build -llspl
```

После этого приложение должно корректно собираться. Для того, чтобы оно запускалось, необходимо, чтобы оно могло найти библиотеку **liblspl.so**. Есть два основных способа это обеспечить:

- Положить библиотеку куда-нибудь в известное программе место (например, в ту же директорию);
- Добавить путь к библиотеке (`<LSPL>/core/build`) в переменную окружения LD_LIBRARY_PATH.

После этого приложение должно корректно запускаться.

Определение шаблонов

Для того, чтобы анализировать текст с помощью шаблонов необходимо сначала определить шаблоны. Для этого в программе необходимо создать два объекта:

- Пространство имен (Namespace), в котором будут храниться скомпилированные шаблоны;
- Построитель шаблонов (PatternBuilder), который осуществляет компиляцию шаблонов во внутреннее представления.

Так уж получилось, что пространство имен уже создано. Необходимо создать построитель шаблонов. Для этого необходимо подключить заголовочный файл:

```
#include <lspl/patterns/PatternBuilder.h>
```

И в код main надо добавить следующую строчку:

```
lspl::patterns::PatternBuilderRef builder = new lspl::patterns::PatternBuilder(ns);
```

И попробовать собрать приложение — все должно быть корректно =).

Теперь, пожалуй стоит обратить внимание на то, что уже второй раз используется не только сам создаваемый класс, но и некоторое имя типа с суффиксом **Ref**. Этот ни что иное, как разделяемый умный указатель — многие объекты Lspl имеют встроенный счетчик ссылок, что позволяет использовать разделяемые указатели и практически не беспокоится об освобождении

памяти. И именно поэтому нигде в тексте программы нет вызова **delete**.

Итак, определение шаблонов. Для того, чтобы определить шаблон необходимо вызвать метод **build** построителя шаблонов и передать туда в качестве аргумента строку, описывающую шаблон:

```
builder->build("NG = {A} N <A=N>");
```

Поскольку в этой строке могут встречаться русские буквы, то необходимо сказать пару слов о кодировке: в библиотеке Lspl по умолчанию используется кодировка CP1251 (это кодировка по умолчанию для русского текста в Windows). Причины тому по большей части исторические, и возможно, что в какой-то момент кодировкой по умолчанию станет UTF8.

Для использования шаблона необходимо подключить заголовочный файл:

```
#include <lspl/patterns/Pattern.h>
```

После того, как шаблон скомпилирован, необходимо получить ссылку на него:

```
lspl::patterns::PatternRef pattern = ns->getPatternByName("NG");
```

Теперь эту ссылку можно будет использовать для наложения на текст.

После подключения шаблонов программа может перестать запускаться — причина в том, что для инициализации модуля морфологии необходимо наличие словарей, которые расположены в `<LSPL>/deps/aot`. Для того, чтобы программа могла их найти, необходимо определить переменную среды RML, указав в ней путь к директории **aot**.

Загрузка и анализ текста

Чтобы анализировать текст, его необходимо сначала загрузить. Для этого необходимо использовать объект класса **PlainTextReader**. Загруженный текст представляется классом **Text**. Сначала необходимо подключить необходимые заголовочные файлы:

```
#include <lspl/text/readers/PlainTextReader.h>
#include <lspl/text/Text.h>
#include <fstream>
```

А потом открыть файл, создать объект загрузчика и загрузить текст в память:

```
  std::ifstream is( "test.txt" );


  lspl::text::readers::PlainTextReader reader;
  lspl::text::TextRef text = reader.readFromStream( is );
```

Опять стоит отметить — текст должен быть в кодировке CP1251. После того, как текст загружен, можно накладывать шаблоны. Опять, необходимо подключить заголовочный файл, в котором определен класс, представляющий наложение:

```
#include <lspl/text/Match.h>
```

Для получения списка наложений шаблона на текста необходимо использовать метод getMatches:

```
lspl::text::MatchList matches = text->getMatches( pattern );
```

После вызова метода **matches** — это вектор ссылок на **Match** и его можно, например, перебирать:

```
for ( uint i = 0; i < matches.size(); ++ i ) {
  lspl::text::MatchRef m = matches[i];
  std::cout << m->getRangeStart() << " " << m->getRangeEnd() << std::endl;
}
```

Здесь **getRangeStart** и **getRangeEnd** возвращают смещения в символах от начала текста начала и конца сопоставленного отрезка.

В принципе, изложенного вполне достаточно, чтобы строить простейшие приложения, осуществляющие анализ текста с использованием Lspl. Дальнейшую информацию о том, что из себя представляют текст и шаблоны можно получить из следующей части.

Текст

Здесь более подробно рассматривается представление в библиотеке текста и некоторые методы работы с ним.

Представление текста

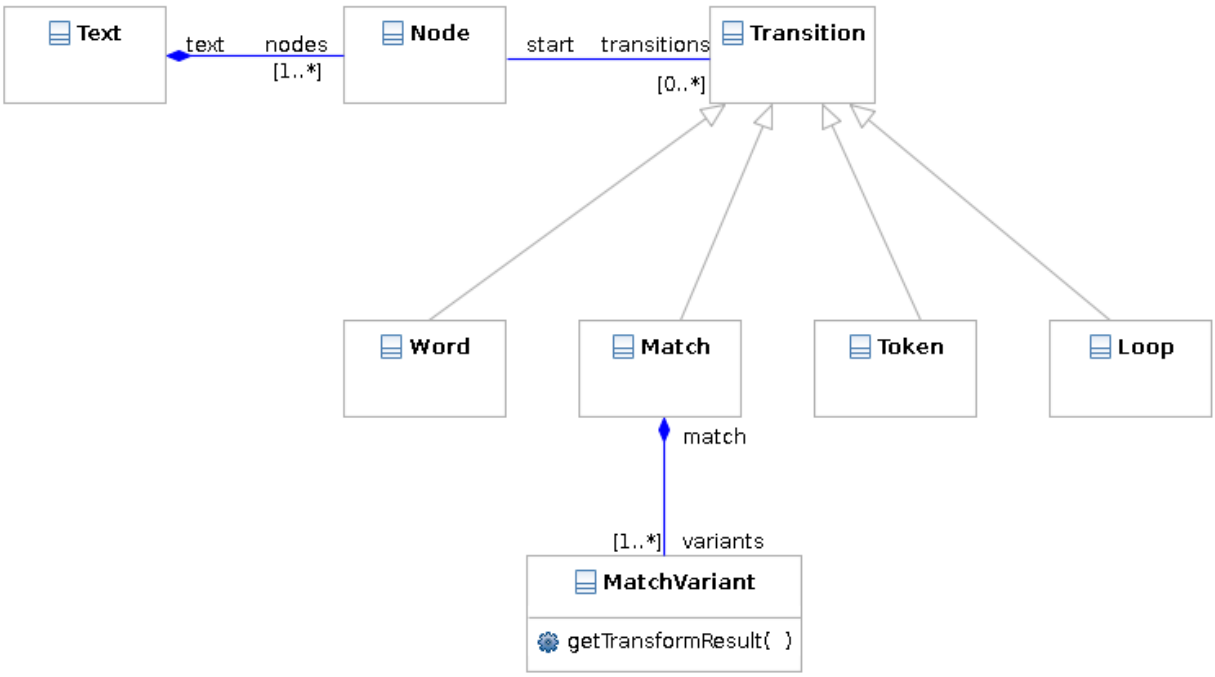
Все классы, используемые в представлении текста находятся в пространстве имен `lspl::text`. Сам текст представляется классом `Text`, который содержит следующие данные:

- Символьная строка (в кодировке CP1251);
- Конфигурация текста, содержащая набор опций с которыми строился текст (режим учета пунктуации, разбиения на предложения и т.п.);
- Граф интерпретаций, содержащий информацию о синтаксических интерпретациях отрезков текста;
- Индексы ребер графа.

И предоставляет следующие методы:

- `getContent()` — получить содержимое текста — строку символов;
- `getWords()` — получить все ребра, соответствующие интерпретациям слов текста. При этом необходимо учитывать, что одно и то же слово может иметь несколько интерпретаций, каждая из которых будет находиться в списке.
- `getWords(SpeechPart)` — получить ребра, соответствующие интерпретациям слов заданной части речи;
- `getMatches(Pattern)` — получить наложения заданного шаблона.

Класс `Text` не предоставляет методов, модифицирующих его содержимое. Это позволяет работать в предположении о немодифицируемости текста, что дает дополнительные возможности для проведения различных оптимизаций.



Вершины графа соответствуют участкам текста, не имеющим никаких синтаксических интерпретаций (например, пробельным символам), и представляются экземплярами класса `Node`, которые могут быть быстро извлечены из текста по номеру. Каждая вершина содержит:

- Ссылку на соответствующий текст;
- Начальное смещение соответствующего незначимого отрезка (в символах);
- Конечное смещение соответствующего незначимого отрезка (в символах);
- Набор ребер, начинающихся в вершине;

И предоставляет методы для получения информации о соответствующем отрезке:

- `getRangeStart()` — получить начальное смещение соответствующего незначимого отрезка;
- `getRangeEnd()` — получить конечное смещение соответствующего незначимого отрезка;
- `getRangeString()` — получить строку, содержащуюся в незначимом отрезке.

Ребра графа соответствуют различным интерпретациям отрезков текста (например, словам в конкретной форме) и представляются объектам различных подклассов абстрактного класса `Transition`, который содержит:

- Ссылки на начальную вершину;
- Ссылку на конечную вершину.

И предоставляет методы:

- `getRangeStart()` — получить начальное смещение соответствующего отрезка;

- `getRangeEnd()` — получить конечное смещение соответствующего отрезка;
- `getRangeString()` — получить строку, содержащуюся в отрезке;
- `getAttribute()` — получить значение заданной характеристики.

Ребра графа

Различные подклассы класса `Transition` представляют различные типы ребер в графе:

- `Token` — неинтепретированный отрезок текста (например, пунктуация);
- `Word` — синтаксическая (морфологическая) интепретация слова;
- `Match` — вариант наложения шаблона на текст.
- `Loop` — наложение цикла на текст.
- `Iteration` — итерация цикла.

Работа с текстом

Здесь описываются некоторые базовые методы работы с текстом в Lspl.

Построение текста

Для загрузки текста во внутреннее представление используются объекты подклассов абстрактного класса `TextReader`, предназначенные для преобразования из различных форматов. Класс содержит два абстрактных метода:

- `readFromString()` — построение текста из строки;
- `readFromStream()` — построение текста из потока.Представление шаблонов

В частности, для построения текста из обычной строки используется класс `PlainTextReader`:

```
lspl::text::readers::PlainTextReader r;


lspl::text::TextRef t1 = r.readFromStream( in ); // Текст из потока
lspl::text::TextRef t2 = r.readFromStream( «Строка» ); // Текст из строки
```

Перебор структуры сопоставления

Хорошим примером перебора ребер, образующих структуру сопоставления может служить класс `lspl::transforms::Normalization` в котором для процесса нормализации используется три основных метода, принимающих ребро, список ребер и список итераций соответственно:

```
void Normalization::appendToString(
  std::string & str,
  const Transition & transition ) const
{
  if ( str.length() > 0 )
```

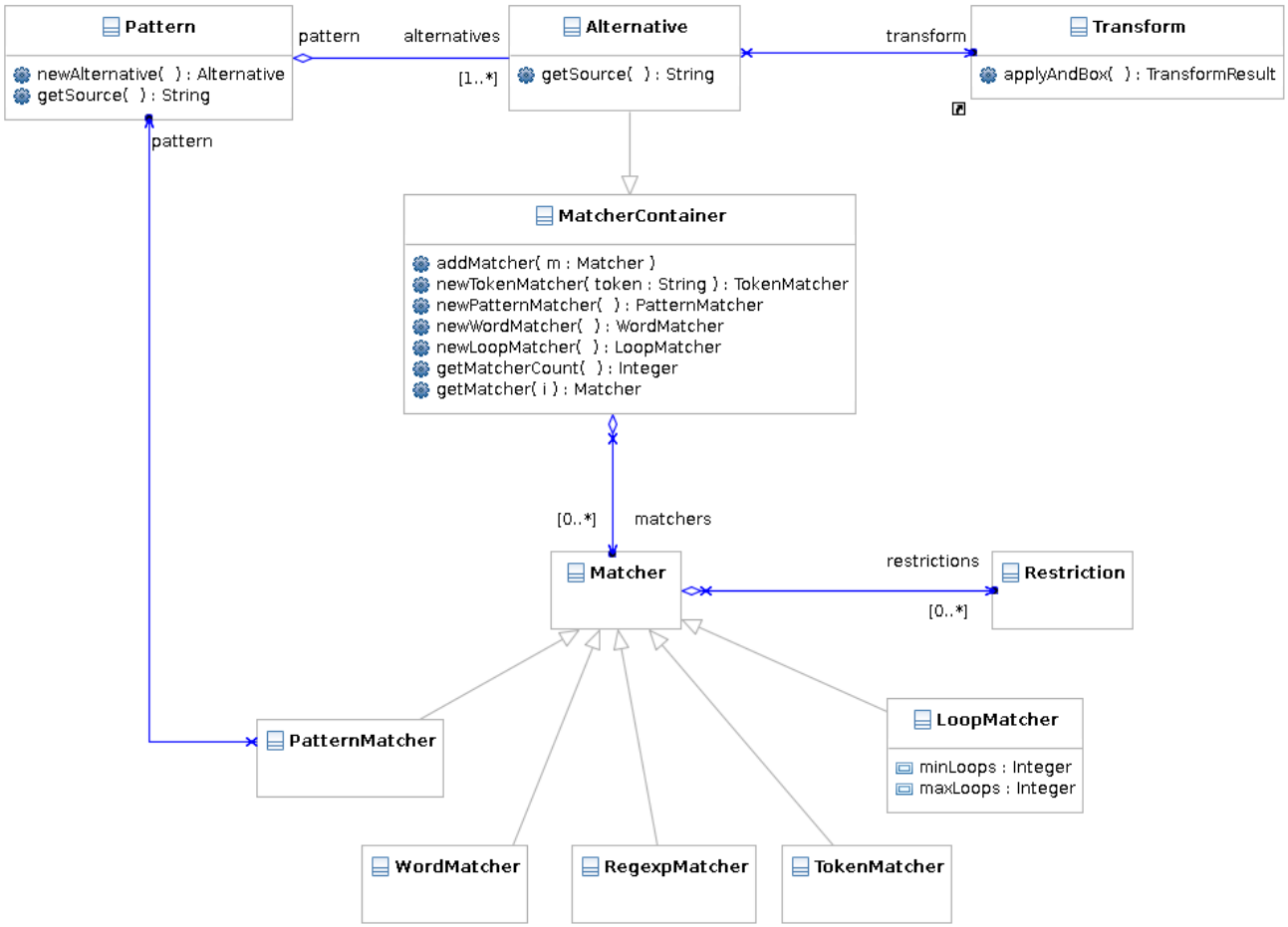
```
      str += " ";

   // Проверяем различные типы ребер и выполняем соответствующие действия
   if ( const Token * token = dynamic_cast<const Token*>( &transition ) ) {
      str += token->getToken();
   } else if ( const Word * word = dynamic_cast<const Word*>( &transition ) ) {
      str += word->getBase();
   } else if ( const Loop * loop = dynamic_cast<const Loop*>( &transition ) ) {
      appendToString( str, loop->getIterations() );
   } else if ( const Match * match = dynamic_cast<const Match*>( &transition ) ) {
      appendToString( str, match->getVariants().at( 0 ) );
   }
}

void Normalization::appendToString(
   std::string & str,
   const TransitionList & transitions ) const
{
   for ( uint i = 0; i < transitions.size(); ++ i ) // Перебираем ребра
      appendToString( str, *transitions[ i ] );
}

void Normalization::appendToString(
  std::string & str,
  const text::LoopIterationList & iterations ) const
{
   for ( uint i = 0; i < iterations.size(); ++ i )
      appendToString( str, iterations[ i ]->getVariant( 0 ) );
}
```

Шаблоны

Здесь рассматривается представление в библиотеке шаблоноы и некоторые методы работы с ними.

Представление шаблонов

Все классы, используемые при описании шаблонов находятся в пространстве имен `lspl::patterns`.



Сам шаблон представляется объектом класса `Pattern`. Из него можно получить имя (`getName()`), исходный код (`getSource()`), набор альтернатив (`getAlternatives()`) и список зависимостей шаблона (`getDependencies()`).

Альтернатива представляется классом `Alternative` и содержит список сопоставителей, представляющих элементы шаблона, связываний с параметрами и преобразования правой части.

Для того, чтобы содержать список сопоставителей класс `Alternative` является наследником класса `matchers::MatcherContainer`, который предоставляет методы для доступа к сопоставителям и создания новых сопоставителей.

Сопоставители

Сопоставители находятся в пространстве имен `lspl::patterns::matchers` и наследуются от класса `Matcher`:

- `TokenMatcher` — сопоставитель, соответствующий строке в кавычках.
- `WordMatcher` — сопоставитель, соответсвующий слову с (возможно) заданными характеристиками.
- `LoopMatcher` — сопоставитель, представлябщий группу элементов в квадратных или фигурных скобках с заданным числом повторений. Он может содержать одну или несколько альтернатив, представленных классом `MatcherContainer` и содержащих

последовательность сопоставителей.

- `PatternMatcher` — сопоставитель, соответствующий другому шаблону.
- `RegexpMatcher` — то же самое, что и `TokenMatcher`, только принимает лексемы, соответствующие регулярному выражению.

Каждому сопоставителю соответсвует некоторый тип ребер, которые они выделяют:

- `WordMatcher` — `Word`
- `TokenMatcher` и `RegexpMatcher` — `Token`
- `PatternMatcher` — `Match`
- `LoopMatcher` — `Match`

Каждый сопоставитель содержит некоторую переменную, с которой соответствующий элемент связан в шаблоне (например, `A1`) и которой он устанавливает значение в процессе сопоставления, а также набор ограничений которые могут ссылаться на переменные, установленные слева от него.

Ограничения

Ограничения используются для проверки различных условий, заданных в шаблоне, например, условий вхождения в словарь или условий согласования. Классы, описывающие ограничения находятся в пространстве `lspl::patterns::restrictions` и являются потомками класса `Restriction`.

Если рассмотреть класс Restriction, то он имеет следующие методы:

- `bool matches(Transition,Context)` — проверка описываемого ограничения на заданном ребре при заданном контексте сопоставления;
- `bool equals(Restriction)` — проверка на равенство заданному ограничению;
- `void dump(std::ostream)` — вывод отладочной формы ограничения.

В настоящий момент существует два возможных типа ограничения (однако планируется их расширить):

- `AgreementRestriction` — проверка согласования, например `<A=N>` или `<A1.c=N.c>`. В процессе работы проверяет согласованность нескольких выражений, вычисленных в рамках контекста сопоставления и текущего кандидата на сопоставленное ребро.
- `DictionaryRestriction` — проверка вхождения в словарь, например `<Syn(N1,N2)>`. При проверке передает значения аргументов классу, реализующему словарь.

Выражения

Выражения находятся в пространстве имен `lspl::patterns::expressions` и унаследованы от класса `Expression`. В настоящий момент существует 5 примитивов, из которых строятся выражения:

- `ConstantExpression` — константа.
- `AttributeExpression` — доступ к аттрибуту

- `VariableExpression` — ссылка на переменную
- `CurrentAnnotationExpression` — ссылка на текущее ребро
- `ConcatenationExpression` — конкатенация

Например, для случая `<A1.c=N1.c>` на сопоставителе `N1` ограничение имеет следующую форму (это не код, хотя он может быть похож):

```
AgreementRestriction(
  AttributeExpression( VariableExpression( A1 ), case ),
  AttributeExpression( CurrentAnnotationExpression(), case )
)
```

Построение шаблонов

Для построения шаблонов используется экземпляр класса `PatternBuilder`, который переводит шаблоны во внутреннее представление. Для этого используется метод `build`, который принимает в качестве аргумента строку в кодировке CP1251. После построения шаблон находится в пространстве имен и может быть получен из него по имени. Например:

```
lspl::NamespaceRef ns = new lspl::Namespace();
lspl::patterns::PatternBuilderRef builder = new lspl::patterns::PatternBuilder(ns);

builder->build("NG = {A} N <A=N>");

lspl::patterns::PatternRef pattern = ns->getPatternByName("NG");
```

Расширенные возможности

Здесь будут описаны расширенные возможности Lspl, такие как использование словарей, прямая работа с морфологией и графематикой, а также преобразование кодировок.

Словари

В LSPL словари представляются не форме множества слов или словосочетаний, а в форме отображения из множества всех словосочетаний в `{True,False}`, то есть по сути в форме характеристической функции этого множества.

Этот подход достаточно удобен с той точки зрения, что позволяет использовать "неявные" словари, например, словарь всех слов, встречающихся в индексах Google или словарь всех буквенных паронимов.

Соответственно, чтобы реализовать словарь необходимо реализовать эту функцию. Для этого необходимо создать класс, реализующий словарь, являющийся потомком класса `lspl::dictionaries::Dictionary` и переопределить в нем метод `acceptsWords`, а в конструктор базового класса передать имя словаря, которое будет использоваться в шаблонах.

В качестве простейшего примера реализации словаря можно использовать `lspl::dictionaries::MemoryDictionary`, который просто хранит множество допустимых словосочетаний в памяти.

Для использования словаря в шаблонах необходимо добавить его в пространство имен (`Namespace`) перед разбором шаблонов. Например:

```
// Создаем пространство имен
NamespaceRef ns = new Namespace();

// Добавляем в него словарь
ns->addDictionary( new MyNewDictionary( "MD" ) );

PatternBuilderRef builder = new PatternBuilder( ns );

// Определяем новый шаблон:
// Два слова, сочетание которых входит в словарь.
builder->build( "TestPattern = A N <MD(A,N)>" );
```

Класс `Dictionary` унаследован от `RefCountObject`, и имеет счетчик ссылок - это надо учитывать при использовании (но использовать макрос `LSPL_REFCOUNT_OBJECT` при объявлении потомков не надо, поскольку `RefCountObject` параметризован классом `Dictionary`, а не потомком).

Морфлогия и графематика

При преобразовании текста во внутреннее представление выполняются процессы графематического и морфологического анализа. Для возможности использования различных анализаторов были выделены абстрактные классы, представляющие модули морфологии (класс `Morphology`) и графематики (класс `Graphan`). В текущей реализации комплекса используются классы `AotMorphology` и `AotGraphan`, основанные на использовании модулей Aot.

Пример использования морфологического модуля для определения форм слова:

```
using lspl::morphology::Morphology;
using lspl::morphology::WordForm;

// Определяем вектор для результатов
boost::ptr_vector<WordForm> forms;

// Вызов морфологического анализа
Morphology::instance().appendWordForms(token->getToken(),forms);

forms[0].getBase() // Это основа слова (его первой интерпретации)
```

Работа с кодировками

TODO

Преобразования результатов (правая часть)

Отдельно стоит описать такую возможность, как преобразование результатов сопоставления. Общая идея состоит в том, что для любой альтернативы шаблона можно описать некоторое правило, которое указывает как из соответствующего этой альтернативе варианта сопоставления получить **нечто**. Что именно? Этот зависит от конкретной задачи и потому не может быть значительно уточнено. Это может быть строка, структура данных, новый шаблон... практически что угодно.

Как это может быть использовано? Например, для перевода результатов сопоставления в некоторые внутренние структуры данных, или же генерации новых шаблонов на основе результатов сопоставления для какой-то итеративной обработки, или вычисления некоторого значения на основе шаблона (например, его веса). Вариантов много, как именно это окажется полезным Вам — зависит от Вашей задачи.

Конечно, за все это приходится чем-то платить — в данном случае тем, что эта преобразование является пожалуй наиболее сложной в использовании возможностью библиотеки и, кроме того, преобразование не дает готового решения конкретной задачи — это скорее место для встраивания такого решения в общий процесс.

Основные понятия

Преобразования результатов основаны на определении и использовании нескольких базовых типов объектов:

- **Результат преобразования** — это то, что получается на выходе преобразования. Это может быть строка, шаблон, или **любая** другая структура данных.
- **Преобразование** — это объект, представляющий функцию, которая принимает в качестве аргумента вариант наложения и вычисляет соответствующий результат преобразования. В общем случае, каждая внешняя альтернатива шаблона имеет собственное, соответствующее ей преобразование.
- **Запись преобразования** — это формальная запись функции преобразования. Она записывается вместе с шаблоном, после символов «=>».
- **Парсер преобразования** — это объект, который осуществляет создание преобразования на основе его записи в шаблоне.

Для того, чтобы использовать механизм преобразований в конкретной задаче необходимо определить объекты, соответствующие этим понятиям, то есть ответить на следующие вопросы:

- Что будет являться результатом преобразования?
- Как будет происходить процесс преобразования из варианта сопоставления в результат? Из каких элементов этот процесс может состоять?
- Как будет формально записываться преобразование? Как будет осуществляться построение объекта, реализующего процесс на основе этой формальной записи?

Представление преобразований

Основные понятия механизма преобразований представлены абстрактными классами в пространстве имен `lspl::transforms`.

Результат преобразования представляется классом `TransformResult`, который позволяет хранить значение любого типа. Это значение может быть извлечено из объекта вызовом метода `getValue` с соответствующей параметризацией.

Преобразование представляется классом `Transform`, который имеет метод `applyAndBox`, принимающий в качестве параметра вариант сопоставления и возвращающий `TransformResult` — то есть запакованный результат преобразования. На практике преобразования стоит наследовать от класса `TypedTransform` с соответствующей параметризацией, который берет на себя процесс запаковки и требует определения только метода `apply`, принимающего тот же самый параметр и возвращающего результат заданного типа.

Парсер преобразования представляется классом `TransformBuilder`, который требует переопределения метода `build`, принимающего в качестве аргументов альтернативу и строку — соответствующую альтернативе запись преобразования.

Использование преобразований

Для того, чтобы использовать механизм преобразований, необходимо выбрать тип результата и создать классы, реализующие преобразования и парсер преобразования.

Затем подключить парсер, необходимо передать его экземпляр вторым параметром в конструктор `PatternBuilder`:

```
lspl::patterns::PatternBuilderRef builder =
  new lspl::patterns::PatternBuilder(ns,new SubstTransformBuilder());
```

После сопоставления результат преобразования может быть получен из варианта наложения вызовом метода `getTransformResult`() с соответствующей параметризацией. Например, в случае перебора результатов наложения при типе результата — строке можно использовать следующий код:

```
for ( uint i = 0; i < matches.size(); ++ i ) {
  lspl::text::MatchRef m = matches[i];


  std::cout << m->getRangeStart() << " "
            << m->getRangeEnd() << " "
            << m->getVariants().at(0).getTransformResult<std::string>()
            << std::endl;
}
```

Пример использования

Например, рассмотрим пример реализации следующего варианта преобразования:

В правой части записывается произвольная строка, в которой встречаются конструкции $1, $2 и так далее до $9. Преобразование есть функция, для каждого варианта наложения возвращающая такую строку (из соответствующей альтернативы), в которой $i заменяется на текст i-го элемента наложения.

В этом случае, типом результата, очевидно, является строка — `std::string`. Класс,

описывающий преобразование может выглядеть следующим образом:

```
class SubstTransform : public lspl::transforms::TypedTransform<std::string>
{
  public:
    SubstTransform(
      const std::vector<std::string> & parts,
      const std::vector<int> & vars ) :
        parts( parts ),
        vars( vars )
    {}

    std::string apply( const lspl::text::MatchVariant & v ) const;
  private :
    std::vector<std::string> parts;
    std::vector<int> vars;
};
```

В нем `parts` — это вектор частей строки между переменными, а `vars` — индексы заменяемых переменных. Предполагается, что верно условие `parts.size() == vars.size() + 1`.

Реализация метода **apply**, которая поочередно склеивает промежутки текста и переменные:

```
std::string SubstTransform::apply(  const lspl::text::MatchVariant & v ) const {
  std::string result = "";

  for ( int i = 0; i < parts.size(); ++ i ) {
    result += parts[i];
    result += v.at( vars[i] - 1 )
      ->getAttribute( lspl::text::attributes::AttributeKey::TEXT ).getString();
  }

  result += parts[ parts.size() - 1 ];

  return result;
}
```

Теперь необходимо определить парсер преобразований, то есть класс, который будет создавать объекты преобразований на основе альтернативы и строки. Например, это можно сделать следующим образом:

```
class SubstTransformBuilder :
  public lspl::transforms::TypedTransformBuilder<std::string>
{
```

```
  public:
    SubstTransformBuilder() {}

    SubstTransform * build( const lspl::patterns::Alternative & alt,
      const std::string & source );
};
```

И реализовать функцию build:

```
SubstTransform * SubstTransformBuilder::build(
  const lspl::patterns::Alternative & alt,
  const std::string & source )
{
  std::vector<std::string> parts;
  std::vector<int> vars;

  int cur = 0;
  int prev = 0;

  std::cout  << source << std::endl;
  while ( std::string::npos != ( cur = source.find( '$', cur ) ) ) {
    std::cout  << cur << std::endl;

     parts.push_back( source.substr( prev, cur - prev ) );
     vars.push_back( source[cur+1] - '0' );

     cur += 2;
     prev = cur;
  }

  parts.push_back( source.substr( prev, source.length() - prev ) );

  return new SubstTransform( parts, vars );
}
```

Теперь все классы готовы, чтобы подключить механизм преобразований, достаточно экземпляр `SubstTransformBuilder` передать вторым аргументом в конструктор класса `PatternBuilder`:

```
lspl::patterns::PatternBuilderRef builder =
  new lspl::patterns::PatternBuilder(ns,new SubstTransformBuilder());
```

И тогда шаблон вида:

```
NG = {A} N <A=N> => NameGroup($2)
```

Может быть преобразован в соответствующую строку.

Аналогичным образом могут быть построены значительно более сложные варианты преобразований.

Использование в Java

TODO