УДК 004.8

А.Е. Ермаков, к.т.н., ведущий специалист по компьютерной лингвистике, ermakov@rco.ru

ЗАО "Крибрум", ООО “ЭР СИ О”, Москва

Метаязык описания грамматики в синтаксических анализаторах естественного языка на основе объектно-ориентированного языка программирования

Аннотация

В статье раскрываются ключевые особенности синтаксического анализатора текста на естественном языке, более 10 лет развиваемого автором в компании "ЭР СИ О", а затем в компании "Крибрум" и ее партнере – компании "Диктум". Описание реализации синтаксического парсера публикуется впервые, однако основное внимание автор хотел акцентировать на новом разработанном метаязыке описания правил формальной грамматики. Назначение метаязыка - описывать правила декларативными средствами языка C++: логическими выражениями и специально реализованными процедурами, тогда как поддержка необходимой процедурной составляющей реализуется интерпретатором метаязыка в ходе синтаксического разбора текста.

Ключевые слова: компьютерный анализ текста на естественном языке, синтаксический анализатор текста, метаязык описания формальной грамматики, дерево синтаксических составляющих, сеть синтактико-семантических отношений.

Введение

Начиная с 2001 года автор занимается разработкой программных компонентов для коммерческих систем лингвистического анализа текста на естественном языке, ядром которых выступает синтаксический анализатор. Зарубежных разработок, осуществляющих разбор русского языка, автору не известно, поэтому речь пойдет только о российских достижениях. Сегодня в России существуют три синтаксических анализатора текстов на русском языке, используемых в коммерческих продуктах: анализаторы компаний ЭР СИ О (*www.rco.ru*), Диктум [2] и ABBYY [3] (перечислены в порядке их появления на рынке). Две другие известные разработки созданы в академических коллективах: синтаксический анализатор ИППИ РАН для системы автоматического перевода ЭТАП-3 [4] и анализатор группы АОТ (*www.aot.ru/docs/synan.html*). В разработке анализаторов ЭР СИ О и Диктума автор принимал непосредственное участие, с работой трех других знаком по результатам общения с их разработчиками и по отзывам клиентов, проводивших независимое тестирование анализаторов для их последующего приобретения. Анализатор ЭР СИ О имеет

производительность в районе 100 мегабайт текста в час, описанный в [2] анализатор Диктума способен обработать в районе 5 мегабайт текста в час, анализаторы ABBYY, ЭТАП-3 и АОТ имеют скорость работы на два порядка ниже, чем у анализатора ЭР СИ О.

Таким образом, статья посвящена описанию реализации самого быстрого анализатора русского текста, принципиально завершенного к 2005 году, но до сих пор нигде не описанному, а также достижению последнего года работы – метаязыку описания правил естественного языка над C++, введение которого позволило устранить ключевой недостаток парсера ЭР СИ О – сложность описания правил грамматики естественного языка, которые раньше писались непосредственно на C++.

Относительно качества синтаксического анализа текста стоит отметить следующее. Полнота и точность разбора определяются не столько подходом к построению парсера, сколько скрупулезностью описания правил грамматики (что определяет полноту) и системой правил выбора наилучшего варианта разбора при омонимии (что определяет точность).

В компании Диктум к 2011 году был разработан синтаксический анализатор, точность работы которого была не высока – на соревнованиях синтаксических парсеров в рамках конференции Диалог-2012 [1] он занял 6-е место, вследствие чего был доработан до состояния, описанного в [2], обеспечив значительно более высокое качество разбора, чем исходный анализатор, и по внутренним тестам разработчиков Диктума (независимое сравнение парсеров на Диалоге больше не проводилось) сравнявшись с анализатором ЭТАП-3, занявшим второе место. Тем не менее, дальнейшее повышение качества на практике обеспечить не удавалось вследствие сложности задания ограничений на недопустимые варианты разбора с точки зрения сочетаемости составляющих в модели [5], в результате чего был осуществлен переход на описываемую в настоящей статье модель синтаксиса и синтаксический парсер, показавшие в компании ЭР СИ О более высокую точность – 95% по результатам тестирования разработчиками (в соревнованиях парсеров на Диалоге-2012 по организационным причинам парсер ЭР СИ О не участвовал). Такая точность сравнима с точностью победителя соревнования по качеству – анализатора ABBYY, хотя достигнутая в ЭР СИ О полнота разбора (процент правильно устанавливаемых синтаксических отношений) несколько уступает полноте других анализаторов и составляет около 90%. Низкая полнота разбора связана с отсутствием в анализаторе правил для множества синтаксических конструкций, относительно редко встречающихся в русскоязычных текстах, вследствие указанного выше недостатка – трудоемкости описания правил непосредственно на языке C++. Именно внедрение описанного в статье метаязыка должно позволить описать более полную систему правил и повысить качество работы наиболее быстрого из созданных в России анализаторов текста.

Модель синтаксиса

Окончательной целью синтаксического анализа является построение структуры связей между словами предложения, которая может быть представлена как сеть направленных и типизированных синтактико-семантических отношений [5]. При этом форма внутреннего представления синтаксической структуры фразы, с которой непосредственно работает синтаксический анализатор, может быть различной, аналогично формам синтаксической разметки предложения лингвистами. Описываемый здесь анализатор использует в своем внутреннем представлении модель деревьев синтаксических составляющих [6,7].

Составляющая соответствует цепочке следующих подряд слов предложения и знаков препинания, она имеет грамматические атрибуты, соответствующие своему вершинному слову – корню дерева составляющей. Исходные слова и знаки препинания являются терминальными составляющими, нетерминальная составляющая покрывает две или более дочерних составляющих (терминальных или нетерминальных), при этом определяя: (а) направленные связи между дочерними составляющими, типы и атрибуты связей, (б) вершинную дочернюю составляющую, каковых может быть несколько, если составляющая покрывает однородные члены. Знание вершинной дочерней составляющей необходимо для установления синтактико-семантических связей между исходными словами, что производится в ходе спуска по дереву составляющих, соответствующему наилучшему варианту разбора предложения. Так, если в составляющей - глагольной группе VP = Verb(*создать*) -> NP*(новые алгоритм и программу*) именная группа NP подчинена глаголу Verb связью типа “управление винительным без предлога”, а в составляющей NP = Adjective(*новые*) <- NP(*алгоритм и программу*) вершинной дочерней составляющей является NP = Noun(*алгоритм*) <-> Noun(*программу*), которая в свою очередь имеет две вершинные составляющие Noun(*алгоритм*) и Noun(*программу*), то в ходе спуска по дереву и связывания вершинных составляющих будут установлены две связи типа “управление винительным без предлога” от *создать* к *алгоритм* и к *программа*. Здесь и далее NP и VP обозначают нетерминальные составляющие, соответствующие именной и глагольной группе, а Noun и Adjective – терминальные составляющие, соответствующие отдельным словам – имени существительному и прилагательному соответственно.

Составляющая в общем случае имеет несколько наборов грамматических атрибутов, каждый из которых характеризует свой омоформ - то слово естественного языка (ЕЯ), грамматической формой которого может быть употребленная в тексте словоформа вершинного слова составляющей. Исключение - составляющие однородных членов предложения, грамматические атрибуты которых формируются аналитически на основе

атрибутов однородных членов (так, грамматическое число у составляющей однородных NP=NP<->NP или VP=VP<->VP всегда будет множественным). Если составляющая нетерминальная, то ей соответствует набор тех омоформов вершинного слова составляющей, которые удовлетворили условиям правил грамматики, задействованных при связывании всех дочерних составляющих в структуру данной - в ходе подъема вверх по дереву составляющих омонимия уменьшается.

Помимо грамматических атрибутов у каждого омоформа, составляющая имеет общие структурные атрибуты, которые описывают структуру дерева ее дочерних составляющих. Например, NP-составляющие (именные группы) могут иметь такие структурные атрибуты: `CONST_ADJ_RIGHT` - прилагательное справа, `CONST_GEN_RIGHT` - несогласованное определение-генитив справа, `CONST_PARTICIPLE_RIGHT` - причастный оборот справа и др. Так, если составляющая s_t имеет грамматической атрибут части речи `POS_NOUN`, а следующая за ней составляющая s_{t+1} имеет грамматические атрибуты `POS_NOUN` и `CASE_GEN` (родительный падеж - генитив), то они не могут входить в составляющую типа NP = NP -> NPgen (именная группа с вершинным существительным в генитиве), если составляющая s_t, имеет хотя бы один из 3-х вышеописанных структурных атрибутов - соответствующее ограничение `CONST_ADJ_RIGHT | CONST_GEN_RIGHT | CONST_PARTICIPLE_RIGHT` записывается в правиле.

Для VP-составляющих (глагольных групп) важнейшими являются такие структурные атрибуты: `CONST_CONTROL_SUBJECT`, `CONST_CONTROL_GEN`, `CONST_CONTROL_DAT`, `CONST_CONTROL_ACC`, `CONST_CONTROL_INSTR`, которые означают, что у вершинного глагола заполнена валентность именительного, родительного, дательного, винительного и творительного падежа без предлога. Знание того, какие валентности VP-составляющей уже заполнены другими NP-составляющими, позволяет включить ограничения, не допускающие возникновения составляющих VP = VP -> NP с многократным заполнением одной валентности.

Синтаксический парсер на C++

Синтаксический парсер обеспечивает применение правил формальной грамматики к последовательности составляющих предложения и построение всех возможных разборов - деревьев составляющих - с выбором наилучшего, без повторного построения возникающих в ходе парсинга поддеревьев. Базовый принцип работы описываемого парсера был почерпнут из работы [8].

На вход парсера поступает последовательность указателей на структуры `SConstituent`, описывающие терминальные составляющие предложения.

```
struct SConstituent {
  long long lConstAttrs; /*структурные атрибуты - 64-битовая маска*/
  vector<SGrammarForm> vGrammar; /*грамматические атрибуты омоформов*/
  vector<SConstituent*> vpChild; /*указатели на дочерние составляющие*/
  vector<SRelationInfo> vRelInfo; /*связи дочерних составляющих*/
};
```

Структура `SGrammarForm` описывает отдельный омоформ в составе поля `vector<SGrammarForm> vGrammar`:

```
struct SGrammarForm {
  long long lGrammar; /*грамматические атрибуты - 64-битовая маска*/
  SGrammarForm* pModifier; /*аналитический модификатор омоформа*/
  SGrammarForm* ppChildG[MAX_RULE_LENGTH]; /*омоформы дочерних составляющих*/
};
```

Здесь `SGrammarForm* pModifier` указывает на другой омоформ - аналитический модификатор грамматической формы данного омоформа (предлог для PP, частица или служебный глагол для VP), значение которого устанавливается в результате работы определенных правил, например PP=Preposition<-NP, VerbAux<-Verb, где PP - именная группа с предлогом, Preposition – предлог, VerbAux – служебный глагол. В массиве `SGrammarForm* ppChildG[MAX_RULE_LENGTH]` сохраняются указатели на омоформы дочерних составляющих, удовлетворившие условиям сформировавшего ее правила, чтобы в финале снять омонимию, спускаясь по дереву разбора и оставляя в дочерних составляющих только те омоформы, которые задействовались в сработавших правилах.

Структура `SRelationInfo` описывает синтактико-семантическую связь отдельной парой дочерних составляющих в составе поля `vector<SRelationInfo> vRelInfo`:

```
struct SRelationInfo {
  SConstituent *pSrc, *pDst; /*указатели на связанные составляющие*/
  int nRelationType; /*тип связи*/
  long long lCase; /*семантический падеж связи*/
  SGrammarForm* pModifier; /*аналогично pModifier в составе SGrammarForm*/
};
```

В составе одной составляющей могут быть установлены связи между несколькими парами дочерних, например, для разбора непроективных конструкций вида "*мозги народу пытаются морочить либералы*" необходимо одно сложное правило бесконтекстной грамматики, формирующее одну составляющую с четырьмя парами связей от вершинной дочерней составляющей *пытаются*.

Результатом разбора является дерево составляющих `SConstituent`, в котором каждый узел хранит указатели `vector<SConstituent*> vpChild` на дочерние узлы.

Все правила грамматики парсера являются C++-классами - наследниками базового класса `CSyntRule` и реализуют его виртуальные методы `Init()`, `_S()`, `G1()`, ... `G5()`, `Concord()`, `Result()`, `Postproc()`. Содержательные особенности реализации этих методов подробно рассмотрены далее. Парсер обращается к каждому из правил, вызывая его методы через указатель на базовый класс `CSyntRule*`, ничего не зная о правиле, кроме длины цепочки обрабатываемых им составляющих `m_nRuleLen`.

Базовый синтаксический парсер работает по следующей схеме.

Двигаясь справа налево по цепочке составляющих, парсер на каждом шаге применяет к каждому фрагменту цепочки каждое правило-реализацию базового класса `CSyntRule`, вызывая его метод `int CSyntRule::Execute( const SConstituent **ppS, SConstituent &NewConstituent )`, с учетом известной для каждого правила длины `m_nRuleLen` покрываемой им цепочки составляющих, не давая правилу выйти за границы предложения. Каждому правилу доступен только переданный ему фрагмент, на который указывают `m_nRuleLen` указателей от `ppS` до `ppS+m_nRuleLen-1` - формальная грамматика является контекстно-свободной. Правило срабатывает, если все его методы `_S()`, `G1()`, ..., `G5()` и `Concord()` возвращают `true` - тогда `CSyntRule::Execute` вызывает метод `CSyntRule::Result()`, который создает единственную результирующую составляющую `NewConstituent`. В новой составляющей `NewConstituent` сработавшим правилом будут заполнены структуры с ее атрибутами, указатели на дочерние составляющие `vpChild`, указатели на все удовлетворившие правилу комбинации омоформов `ppChildG` для последующего снятия омонимии.

Будем называть шагом разбора S^r_t применение правила с номером r (r=1..R, где R – количество правил) к фрагменту последовательности составляющих F^r_t с позиции t (t=1..T) до позиции `t+m_nRuleLen-1` правила r. Парсинг начинается с t=T, где T – длина последовательности составляющих верхнего уровня, указатели на которые хранятся в массиве `SConstituent **ppS`.

Если на очередном шаге S^r_t виртуальный метод `CSyntRule::Execute` правила r вернул 0, то парсер пытается применить правило r+1 к позиции t. Если все правила вплоть до R для позиции t вернули 0, то парсер пытается применить правило r=1 к позиции t-1 и т.д., пока не сработает какое-то правило - `CSyntRule::Execute` вернет 1.

Если на очередном шаге S^r_t `CSyntRule::Execute` вернул 1, то цепочка указателей F^r_t заменяется указателем на новую покрывающую составляющую `NewConstituent` - цепочка дочерних составляющих сворачивается. Весь набор правил применятся сначала с той же

позиции t (теперь этот фрагмент содержит новую составляющую `NewConstituent`) – проверяется правило r=1 и т.д.

Информация о произведенных заменах составляющих сохраняется в т.н. протоколе разбора - логическом стеке, каждый элемент которого хранит указатель на указатель на новую вставленную составляющую `NewConstituent` в массиве `SConstituent **ppS`, номер составляющей в последовательности t и номер правила r, ее сформировавшего.

На последнем шаге S^R_1 парсер завершает построение очередного варианта разбора, после чего производится:

- сравнение текущего разбора и лучшего из ранее построенных по набору критериев. Если текущий разбор оказывается лучше, то полученное дерево составляющих сохраняется взамен старого.

- откат состояния разбора – из протокола выбирается указатель на последнюю построенную покрывающую составляющую с соответствующими ей t и r, на основании чего восстанавливается предыдущее состояние разбора – на свое место в массиве `SConstituent **ppS` вставляются сохраненные в покрывающей составляющей указатели на дочерние составляющие, сама покрывающая и указатель на нее удаляются, а парсер переходит к шагу S^{r+1}_t если r<R, или S^1_{t-1} если r=R и t>1.

Если стек протокола пуст и выполнен шаг S^R_1, разбор заканчивается.

Описанный парсер иллюстрирует базовый принцип построения всех деревьев разбора, допустимых в рамках используемого множества правил грамматики. Однако он не является оптимальным с точки зрения производительности, поскольку не учитывает того факта, что одно и то же дерево составляющих может быть собрано из своих поддеревьев в разном порядке - в описанной логике уникальность составляющей определяется только порядком ее сборки, и все одинаковые по структуре составляющие будут далее обрабатываться правилами как разные, порождая формально новые, а фактически эквивалентные деревья разбора. Поэтому на практике используется более сложная реализация парсера, которая хранит информацию о ранее обработанных соседях составляющих, уже возникавших в ходе разбора, во избежание повторного парсинга.

Метаязык описания правил грамматики на C++

Настройка правил формальной грамматики является наиболее ресурсоемкой задачей при создании синтаксического анализатора ЕЯ, и основными ее исполнителями являются лингвисты, а не программисты. Потому наиболее практически приемлемой кажется такая реализация, в которой правила описываются не на языке программирования (ЯП), а на неком

формальном декларативном языке, как, например, реализация, выполненная в компании Диктум [1]. Однако подобные реализации имеют два недостатка:

- чтобы использовать правила при разборе, необходим интерпретатор этих правил, в простейшем случае - непосредственно из их исходного текста, а в наилучшем - из промежуточного бинарного представления, сформированного специальной программой - компилятором с исходного языка, и хранящего данные о правилах в специальных структурах для быстрого обращения к ним в ходе разбора. Так или иначе, производительность оказывается более низкой, чем при написании правил непосредственно на том ЯП, на котором написан синтаксический парсер.

- среди порождаемых конструкций ЕЯ находятся исключения, которые не удается описать средствами декларативного языка формальной грамматики. В то же время возможности разумного расширения выразительных средств декларативного языка не безграничны и многие ситуации оказывается проще обрабатывать процедурно средствами ЯП, не расширяя всякий раз формализм декларативного языка.

Правильный подход к написанию правил на универсальном ЯП позволяет избавиться от обоих недостатков. ЯП поддерживает все логические операторы и позволяет описывать любые логические выражения для проверки на возможную сочетаемость грамматических и структурных атрибутов составляющих. Чтобы избежать "человечески нечитаемых" описаний грамматики из-за включения в правила процедурной логики (условные переходы, циклы, присваивания переменных) и допустить к написанию правил лингвиста, незнакомого с программированием, необходимо определить стандарт написания правил - декларативный метаязык над ЯП, и разработать его интерпретатор, который априорно будет быстрее, чем любой интерпретатор внешнего к ЯП декларативного языка. При этом сохраняется возможность использовать все средства процедурного программирования для написания нестандартных правил в исключительных случаях.

Разработанный метаязык построен над C++, его основные элементы следующие:

- `s1`, `s2`, `s3`, `s4`, `s5` - порядковые обозначения пяти составляющих, к которым могут применяться операции в правилах. Для русского языка самое длинное правило, задействующее пять составляющих, необходимо для покрытия непроективной конструкции вида "*мозги народу пытаются морочить либералы*", поскольку описание соответствующей непроективной конфигурации синтаксических связей, устанавливаемых от четвертого слова к первому и второму, а также от пятого к третьему, должно производиться в рамках единого правила. Теоретически в языке подобная конструкция может содержать и более длинную цепочку инфинитных глаголов, например "*мозги народу пытаются начать морочить*

либералы", что соответствует правилу для шести составляющих, однако практически кажется разумным ограничиться пятью.

- `CONST(s,ValueMask)` - предикат сравнения структурных атрибутов составляющей `s` со значением выражения `ValueMask`, например: `CONST(s2, CONST_ADJ_RIGHT|CONST_PP_RIGHT)` имеет значение `true`, если у составляющей `s2` есть атрибут "включает прилагательное справа" или "включает PP справа".

- `GRAM(s,ValueMask)` - предикат сравнения грамматических атрибутов составляющей `s` со значением выражения `ValueMask`, например: `GRAM(s2,POS_NOUN|POS_PRONOUN)` имеет значение `true`, если у составляющей `s2` проверяемый правилом омоформ - это существительное или местоимение. Специальные предикаты `gPLURAL(s)`, `gSINGULAR(s)` принимают значение `true` в зависимости от грамматического числа омоформа.

- `gCASE(s1,s2)`, `gGENDER(s1,s2)`, `gPERSON(s1,s2)` - предикаты проверки согласования, которые возвращают `true` в случае совпадения значений грамматических атрибутов у сопоставляемых правилом омоформов из двух заданных составляющих `s1` и `s2`. Имя предиката определяет проверяемый на согласование атрибут, здесь указаны предикаты для проверки падежа, рода и лица. - `VALENCE(s1,s2)` - предикат проверки управления - имеет значение `true`, если для пары сопоставляемых правилом омоформов из `s1` и `s2` второй омоформ удовлетворяет модели управления первого - имеет допустимый падеж и предлог.

- `SetConstituent(ValueMask)`, `SetGrammar(ValueMask)` - установка наборов значений `ValueMask` структурных и грамматических атрибутов составляющей, покрывающей цепочку дочерних составляющих, удовлетворяющих правилу. `SetGrammar(s)` - установка всех грамматических атрибутов покрывающей составляющей из обрабатываемого правилом омоформа дочерней составляющей `s`.

- `SetRelation(s1,s2,RelationType,Case,Modifier)` - установка направленной связи типа `RelationType` с семантическим падежом `Case` и модификатором `Modifier` (предлогом, союзом) между вершинными дочерними составляющими в составляющих `s1` и `s2`. Падеж и модификатор задаются только для связей определенных типов, обычно устанавливаемых на основе моделей управления предикатов [1].

Рассмотрим теперь структуру правила грамматики, которое представляет собой C++-класс, наследованный от базового `CSyntRule` и реализующий его виртуальные методы с использованием вышеописанных элементов метаязыка.

```
class CSR_Adjective_NP : public CSyntRule {
  virtual void Init(){ _C( 0, 1 ); _R(s2, s1, REL_ATTRIBUTE); }
```

```
  virtual bool _S() { return !CONST(s2,CONST_ADJ_RIGHT|CONST_PP_RIGHT); }
  virtual bool G1() { return GRAM(s1, POS_ADJ); }
  virtual bool G2() { return GRAM(s2, POS_NOUN|POS_PRONOUN); }
  virtual bool Concord() { return gCASE(s1,s2) && ((gPLURAL(s1) && gPLURAL(s2))
|| (gGENDER(s1, s2) && gSINGULAR(s1) && gSINGULAR(s2)));
  }
  virtual void Result() {
    SetConstituent(GetConstituent(s2) | CONST_NP|CONST_ADJ_LEFT);
    SetGrammar(s2);
  }
};
```

Данное правило имеет имя `CSR_Adjective_NP` и обрабатывает составляющую со структурой NP=Adjective<-NP - прилагательное, подчиненное существительному в вершине другой NP.

Метод `Init()` задает длину цепочки дочерних составляющих и помечает вершинную в цепочке, а также определяет отношения между терминальными составляющими предложения после завершения его синтаксического разбора на основе полученного дерева составляющих. Здесь `_C(0,1)` указывает, что правило обрабатывает две составляющие (Adjective и NP), из которых вторая является вершинной; а `_R(s2,s1,REL_ATTRIBUTE)` предписывает установить связь типа `REL_ATTRIBUTE` от каждой вершинной составляющей из `s2` (каковых может быть несколько, если `s2` - группа однородных NP) к каждой вершинной составляющей из группы `s1` (каковых может быть несколько, если `s1` - группа однородных Adjective).

Метод `_S()` вызывается интерпретатором метаязыка для проверки ограничения на структурные атрибуты составляющих, обрабатываемых правилом. Здесь `!CONST(s2, CONST_ADJ_RIGHT|CONST_PP_RIGHT)` означает, что для `s2` (NP) недопустимо иметь прилагательное справа от существительного - по стилистике, если таковое уже имеется справа, то ненормально подчинять второе прилагательное слева. `CONST_PP_RIGHT` - запрет на включение PP в состав NP, связанный с ограничениями на допустимый порядок сборки составляющих при парсинге во избежание сборки одной и той же составляющей разными путями: NP=(Adjective<-NP)->PP и NP=Adjective<-(NP->PP), ограничение `CONST_PP_RIGHT` запрещает второй порядок сборки.

Методы `G1()-G5()` вызываются интерпретатором метаязыка для независимой проверки ограничений на грамматические атрибуты омоформов составляющих `s1-s5` без проверки их согласования. Эти проверки также могут быть выполнены непосредственно в

методе Concord() (тогда не требуется имплементация G1()-G5()), но тогда, как можно увидеть из реализации интерпретатора далее, скорость его работы снизится.

Метод Concord() вызывается для полной проверки ограничений на грамматические атрибуты с согласованием их значений между омоформами разных составляющих. Проверка ограничений производится интерпретатором для всех комбинаций омоформов из K<=5 составляющих, обрабатываемых правилом, в ходе работы K вложенных циклов (см. реализацию CSyntRule::Execute() далее). Вынос проверок независимых ограничений в методы G1()-G5() позволяет во многих случаях избежать прохода во вложенные циклы.

Метод Result() вызывается интерпретатором для создания покрывающей составляющей для цепочки дочерних, удовлетворившей ограничениям, описанным в методах _S(), G1()-G5(), Concord(). Result() вызывается для каждой комбинации омоформов цепочки, удовлетворяющей ограничениям, и всякий раз в покрывающую составляющую добавляется новый омоформ с атрибутами, устанавливаемыми посредством SetGrammar. SetConstituent(GetConstituent(s2) | CONST_NP|CONST_ADJ_LEFT) задает структурные атрибуты покрывающей составляющей как копию атрибутов составляющей s2, плюс атрибуты CONST_NP (NP-составляющая) и CONST_ADJ_LEFT (содержит прилагательное слева), а SetGrammar(s2) устанавливает грамматические атрибуты омоформа покрывающей составляющей от омоформа s2, удовлетворившего всем ограничениям правила.

Ниже приведен пример правила CSR_Homogenous для составляющей из однородных членов: именных групп NP, прилагательных или наречий, перечисленных через ',' или сочинительный союз POS_CONJ_COORD, что проверяется для составляющей s2 в G2(). _C(1,0,1) показывает, что вершинными являются две равноправные составляющие s1 и s3. Проверка !CONST(s1,CONST_HOMOGENOUS) позволяет сборку составляющих с более чем двумя однородными членами только в одном направлении "справа налево". В методе Concord() gPOS(s1, s3) проверяет совпадение частей речи омоформов составляющих s1 и s3, а затем, если часть речи омоформов прилагательное или существительное, производится проверка согласования падежей - gCASE(s1, s3). В методе Result() покрывающая составляющая получает структурные атрибуты обеих дочерних s1 и s3, плюс атрибут CONST_HOMOGENOUS. Омоформ покрывающей получает грамматические атрибуты омофрма s1, плюс атрибут множественного числа NUMBER_PLURAL. В методе Init() предписывается установить связь типа REL_HOMOGENOUS между дочерними вершинными составляющими составляющих s1 и s3 симметрично в обе стороны.

```
class CSR_Homogenous : public CSyntRule {
  virtual void Init() { _C(1, 0, 1);
```

```
    _R(s3, s1, REL_HOMOGENOUS); _R(s1, s3, REL_HOMOGENOUS);
  }
  virtual bool _S() { return !CONST(s1, CONST_HOMOGENOUS); }
  virtual bool G2() { return GRAM(s2, POS_CONJ_COORD) || TEXT(s2, L","); }
  virtual bool Concord() {
    return gPOS(s1, s3) && (GRAM(s1, POS_ADVERB) || ((GRAM(s1, POS_ADJ) ||
GRAM(s1, POS_NOUN)) && gCASE(s1, s3)));
  }
  virtual void Result() {
    SetConstituent(GetConstituent(s1) | GetConstituent(s3) | CONST_HOMOGENOUS);
    SetGrammar(s1); SetGrammar(NUMBER_PLURAL);
  }
};
```

Ниже пример особого базового правила `CSR_VerbControl` для составляющей вида VP=VP->NP, в которой вершина VP - глагол или предикатив - подчиняет себе именную группу через управление ее падежом. Правило параметризовано значением обрабатываемого падежа и реализует базовый C++-класс для реализации правил под конкретные падежи через механизм наследования, для чего имеет конструктор с параметрами `CSR_VerbControl( long long Case, long long Constituent)`, которые задают обрабатываемый падеж и структурные атрибуты покрывающей составляющей. Предикат `VALENCE(s1,s2)` проверяет возможность падежного управления омоформа из `s1` омоформом из `s2`.

```
class CSR_VerbControl: public CSyntRule {
  long long lCase, lConstituent;
protected:
  CSR_VerbControl(long long Case, long long Constituent): CSyntRule(),
lCase(Case), lConstituent(Constituent) {}

  virtual void Init() { _C(1, 0); _Relation(s1, s2, REL_CONTROL, gCASE(s2)); }
  virtual bool _S() {
    return !CONST(s1, CONST_S_SUBJECT | lConstituent) && !CONST(s2, CONST_PP );
  }
  virtual bool G1() { return GRAM(s1, POS_VERB|POS_PREDICATIVE); }
  virtual bool G2() { return GRAM(s2, POS_COM_NOUN) && GRAM(s2, lCase); }
  virtual bool Concord() { return VALENCE(s1, s2); }
  virtual void Result() {
    SetConstituent(GetConstituent(s1) | lConstituent | CONST_VP|CONST_NP_RIGHT);
    SetGrammar(s1);
  }
};
```

Реальные правила для составляющих VP=VP->NP с управлением конкретными падежами тривиально реализуются через наследование от правила `CSR_VerbControl`, как в примере ниже для дательного падежа:

```
class CSR_VerbControl_Dat : public CSR_VerbControl {
public:
  CSR_VerbControl_Dat() : CSR_VerbControl(CASE_DAT, CONST_CONTROL_DAT){}
};
```

Интерпретатор метаязыка

Интерпретатор метаязыка реализуется в классе `CSyntRule`, ключевой программный код которого представлен ниже.

```
class CSyntRule {
protected:
  SConstProcInfo s1, s2, s3, s4, s5; /*данные об обработке составляющих*/
  SConstituent* pS; /*указатель на результирующую покрывающую составляющую*/
  SGrammarForm* pG; /*атрибуты очередного омоформа покрывающей составляющей*/
  /*маска признаков дочерних составляющих: 1 - составляющая вершинная*/
  unsigned m_vnConstMask[MAX_RULE_LENGTH];

public:
  int m_nRuleLen; /*количество составляющих, покрываемых правилом*/

  int Execute( const SConstituent** ppS, SConstituent &NewConstituent )
  {
    switch( m_nRuleLen ) {
    case 2: s1.pS=ppS[0], s2.pS=ppS[1], s3.pS=NULL, s4.pS=NULL;
    case 3: s1.pS=ppS[0], s2.pS=ppS[1], s3.pS=ppS[2], s4.pS=NULL;
    ...
    };

    if( !_S() ) /*условия на структурные атрибуты составляющих не выполнены*/
      return 0;
    pS = &NewConstituent;
    bool bRuleFired = false;
    for(s1.iG=s1.pS->vGrammar.begin(); s1.iG!=s1.pS->vGrammar.end(); ++s1.iG)
    if( G1() ) /*условие на атрибуты очередного омоформа s1.iG*/
      for(s2.iG=s2.pS->vGrammar.begin(); s2.iG!=s2.pS->vGrammar.end(); ++s2.iG)
      if( G2() ) /*условие на атрибуты очередного омоформа s2.iG*/
        if( m_nRuleLen == 2 )
        {
          if( FireRule() ) bRuleFired = true;
        }
```

```
        else
        for(s3.iG=s3.pS->vGrammar.begin();s3.iG!=s3.pS->vGrammar.end();++s3.iG)
        if( G3() ) /*условие на атрибуты очередного омоформа s3.iG*/
          if( m_nRuleLen == 3 )
          {
            if( FireRule() ) bRuleFired = true;
          }
          else
          ... /*аналогично циклы проходят по омоформам s4 и s5*/
                if( G5() )
                {
                  if( FireRule() ) bRuleFired = true;
                }
    return bRuleFired;
  }

  bool FireRule()
  {
    if( !Concord() ) /*условия на согласование омоформов не выполнены*/
      return false;
    pS->m_vGF.push_back( SGrammarForm() );
    pG=&(pS->m_vGF.back());
    pS->vpChild[0]=s1.pS, pG->ppChildG[0]=s1.iG.operator->();
    if( m_nRuleLen>1 ) pS->vpChild[1]=s2.pS, pG->ppChildG[1]=s2.iG.operator->();
    if( m_nRuleLen>2 ) pS->vpChild[2]=s3.pS, pG->ppChildG[2]=s3.iG.operator->();
    ...
    Result(); /*установить атрибуты покрывающей составляющей*/
    return true;
  }

protected:
  /*указание покрываемых правилом составляющих и маркировка вершинных*/
  void _C( int S1, int S2 ){ m_nRuleLen=2; m_vnConstMask[0]=S1,
m_vnConstMask[1]=S2; }
  void _C( int S1, int S2, int S3 ){ m_nRuleLen=3; ... }
  void _C( int S1, int S2, int S3, int S4 ){ m_nRuleLen=4; ... }
  void _C( int S1, int S2, int S3, int S4, int S5 ){ m_nRuleLen=5; ... }

  /*виртуальные методы для имплементации в правилах-наследниках CSyntRule*/
  virtual void Init() = 0;
  virtual bool _S() { return true; }
  virtual bool G1() { return true; }
  ...
```

```
  virtual bool G5() { return true; }
  virtual bool Concord() { return true; }
  virtual void Result() = 0;
  virtual void Postproc() = 0;

  /*предикаты и прочие элементы метаязыка*/
  bool CONST( const SConstProcInfo &s, long long lConstAttrs ) {
    return s.pS->lConstAttrs & lConstAttrs );
  }
  bool GRAM( const SConstProcInfo &s, long long lGramAttrs ) {
    return s.pG->lGrammar & lGramAttrs;
  }
  ...
};
```

Структура `SConstProcInfo` хранит информацию об обработке составляющей в ходе выполнения правила:

```
struct SConstProcInfo {
  SConstituent* pS; /*указатель на составляющую*/
  vector<SGrammarForm>::iterator iG; /*указатель на обрабатываемый омоформ*/
};
```

Через объекты `s1-s5` типа `SConstProcInfo`, указываемые в качестве аргументов предикатов и специальных методов метаязыка в правилах грамматики, в интерпретаторе производится адресация к атрибутам составляющих и их омоформов: см. реализацию предикатов `CONST` и `GRAM` в теле класса `CSyntRule`.

Работа интерпретатора выполняется в методе `CSyntRule::Execute(const SConstituent** ppS, SConstituent &NewConstituent)`, который является общим методом всех классов-наследников, реализующих конкретные правила грамматики, и вызывается синтаксическим парсером для проверки условий на `m_nRuleLen` составляющих предложения, начиная с той, на которую указывает аргумент `ppS`. В случае удовлетворительной проверки всех условий правила `CSyntRule::Execute` возвращает парсеру 1, а в `NewConstituent` формируется покрывающая составляющая для обработанной цепочки дочерних.

Для проверки ограничений правила на всех комбинациях омоформов из `m_nRuleLen=K` обрабатываемых составляющих `s1-sK` указатели `iG` пробегают все омоформы в ходе `K` вложенных циклов, и для каждого омоформа `iG` выполняется проверка условий `G1()-GK()`, а для каждой комбинации этих омоформов, успешно прошедших проверки условий `G1()-GK()`, выполняется проверка условий на согласование их грамматических атрибутов `Concord()`. Такой подход позволяет скрыть за синтаксисом метаязыка всю процедурную

составляющую работы правил грамматики: циклы по омоформам, условные переходы при обработке ограничений на атрибуты составляющих, операции с указателями C++.

Метод `CSyntRule::FireRule()` вызывается из метода `CSyntRule::Execute` для каждой текущей комбинации омоформов, прошедших проверку независимых ограничений на грамматические атрибуты `G1()-G5()`, и выполняет проверку условий согласования атрибутов вызовом виртуального метода `Concord()`, реализованного в правиле-наследнике. Если все условия соблюдены, то далее над покрывающей составляющей по указателю `pS` производятся следующие действия:

- в массиве `pS->vpChild` сохраняются указатели на дочерние составляющие;
- в массив `pS->vGrammar` добавляется очередной "пустой" омоформ для последующей простановки его атрибутов;
- вызывается виртуальный метод `Result()`, реализованный в правиле-наследнике, который проставляет структурные атрибуты покрывающей составляющей и грамматические атрибуты ее очередного омоформа;
- в массиве `pG->ppChildG` сохраняются указатели на текущие омоформы дочерних составляющих как подтвердившиеся правилом грамматики для последующего снятия омонимии на основе дерева наилучшего разбора.

Заключение

Описанный метаязык описания грамматики ЕЯ на C++, его интерпретатор и синтаксический парсер послужили основой для разработки новой версии лингвистического анализатора русского текста в компании "Диктум" (*www.dictum.ru*). Лингвистический анализатор используется в электронных сервисах мониторинга информации в социальных сетях, предоставляемых системой "Крибрум" (*www.kribrum.ru*), для выявления позитивных/негативных оценок объектов мониторинга, сбора фактов по объектам мониторинга и авторам сообщений, а также в компании “ЭР СИ О” (*www.rco.ru*) для решения аналогичных задач.

Литература

1. Толдова С. Ю., Соколова Е. Г., Астафьева И. и др. Оценка методов автоматического анализа текста 2011–2012: синтаксические парсеры русского языка // Компьютерная лингвистика и интеллектуальные технологии: По материалам ежегодной Международной конференции «Диалог» (Бекасово, 30 мая–3 июня 2012 г.). Вып. 11 (18): В 2 т. Т. 2: Доклады специальных секций – М.: Изд-во РГГУ, 2012. – С. 77-90.

2. Skatov D.S., Liverko S.V., Okatiev V.V., Strebkov D.Y. (2013), Parsing Russian: a Hybrid Approach, Association for Computational Linguistics (ACL), Proceedings of the 4th Biennial International Workshop on Balto-Slavic Natural Language Processing.
3. Anisimovich K. V., Druzhkin K. Ju., Minlos F. R. et al. Syntactic and semantic parser based on ABBYY Compreno linguistic technologies // Компьютерная лингвистика и интеллектуальные технологии: По материалам ежегодной Международной конференции «Диалог» (Бекасово, 30 мая–3 июня 2012 г.). Вып. 11 (18): В 2 т. Т. 2: Доклады специальных секций – М.: Изд-во РГГУ, 2012. – С. 91-103.
4. Iomdin L., Petrochenkov V., Sizov V., Tsinman L. ETAP parser: state of the art // Компьютерная лингвистика и интеллектуальные технологии: По материалам ежегодной Международной конференции «Диалог» (Бекасово, 30 мая–3 июня 2012 г.). Вып. 11 (18): В 2 т. Т. 2: Доклады специальных секций – М.: Изд-во РГГУ, 2012. – С. 119-131.
5. Ермаков А.Е. Эксплицирование элементов смысла текста средствами синтаксического анализа-синтеза. // Компьютерная лингвистика и интеллектуальные технологии: труды Международной конференции Диалог’2003. – Москва, Наука, 2003. - С. 136-140.
6. Гладкий А.В. Синтаксические структуры естественного языка в автоматизированных системах общения. М.: Наука, 1985. - 144 с.
7. Тестелец Я.Г. Введение в общий синтаксис. М.: Издательство РГГУ, 2001. — 798 с.
8. Ю. М. Смирнов, А. М. Андреев, Д. В. Березкин, А. В. Брик. Об одном способе построения синтаксического анализатора текстов на естественном языке // Известия вузов. Приборостроение, 1997. Т. 40, № 5 — стр. 34—42.