АВТОМАТИЗАЦИЯ ОТЛАДКИ АЛГОРИТМОВ ПОВЕРХНОСТНО-СИНТАКСИЧЕСКОГО АНАЛИЗА

А.М. Баталина

batalina_anna@rambler.ru

Г.Ю. Айриян

jorge@mail.ru

М.Е. Епифанов

eme@plms.phys.msu.ru

Т.Ю. Кобзарева

stamstam@mtu-net.ru

Д.Г. Лахути

delir1@yandex.ru

РГГУ, Москва

Продолжено описание системы объектного моделирования поверхностно-синтаксического анализа (Диалог'2004). Рассмотрена конкретная реализация пошагового выполнения алгоритмов поверхностно-синтаксического анализа – аналог режима отладки (Debug) в инструментальных средах разработки программ.

Введение

Любая система автоматической обработки текста использует те или иные правила. При подходе к задаче поверхностно-синтаксического анализа (ПСА), развиваемом в [1-3], правила организуются в так называемые алгоритмы, являющиеся аналогом блок-схем. Правила связаны между собой так, что от одного правила можно переходить не более чем к двум другим: переход к одному из них осуществляется в зависимости от выполнения или невыполнения условий правила. Само правило содержит условия, применяемые к анализируемому предложению или некоторой его части и, возможно, действия, модифицирующие представление предложения, если условия правила выполняются.

Такая организация правил в виде алгоритма, с одной стороны, наглядно изображает процесс ПСА. С другой стороны, она удобна своей процедурностью (алгоритм, по сути, является процедурой, при проходе которой осуществляется некоторая часть ПСА). Последнее свойство существенно облегчает программную реализацию ПСА.

Важной чертой системы алгоритмов является степень ее отлаженности. До сих пор процесс отладки специалисту приходилось проводить, условно говоря, «вручную», а сами алгоритмы записывались и сохранялись в виде набора документов MS Word. Это значительно затрудняло процесс разработки новых правил и их дальнейшей организации, поскольку их число уже достаточно велико и умозрительно экспериментировать со всей совокупностью алгоритмов достаточно сложно.

При разработке правил и их эффективной организации в алгоритмы желательно иметь возможность отслеживать детали происходящего в процессе их применения, т.е. возможность их детального тестирования и отладки. Кроме того хотелось бы поддерживать справочную информацию о корпусе алгоритмов и примерах к ним, о вариантах алгоритмов и правил. И чем богаче такого рода инструментарий, тем эффективнее может быть проведена доработка и отладка корпуса алгоритмов.

Эти задачи призвана решать инструментальная среда для экспериментов с алгоритмами поверхностно-синтаксического анализа [4], реализуемая в Отделении интеллектуальных систем (в гуманитарной сфере) Института лингвистики РГГУ. В данной работе мы продолжаем ее описание, акцентируя внимание на средствах отладки алгоритмов ПСА.

Объектное представление алгоритмов поверхностно-синтаксического анализа

Рассматриваемая инструментальная среда базируется на системе объектного моделирования

поверхностно-синтаксического анализа [5]. Данная модель представляет собой расширение объектной модели многофункциональных словарей, основанной на синтезе лингвистических единиц [6]. В ее основе лежит идея многоуровневого представления лексических единиц вместе с их свойствами в виде иерархии объектов, объединенных в динамическую многоссылочную структуру. Для моделирования поверхностно-синтаксического анализа добавляются объекты, представляющие алгоритмы, их узлы, правила.

Поясним схему объектного представления алгоритма в модели (рис.1). Алгоритм представляется в виде сети объектов. Корневой объект представляет алгоритм в целом. Он ссылается на единственный начальный узел сети правил ("узел 1"). Каждый из узлов сети правил описывает применение некоторого правила в контексте алгоритма и имеет следующую структуру: ссылка на представляющий правило объект, ссылки на дочерние узлы (не более двух). Вообще говоря, на одно правило может ссылаться несколько узлов, возможно, из разных алгоритмов. Правило, в свою очередь, хранит свои внутренние элементы, среди которых – присваивания значений использующимся в нем (и в его потомках) переменным, условия и действия. Алгоритмы не обязательно "изолированы" друг от друга, одни алгоритмы могут вызываться из других – аналогично процедурам в языках программирования.

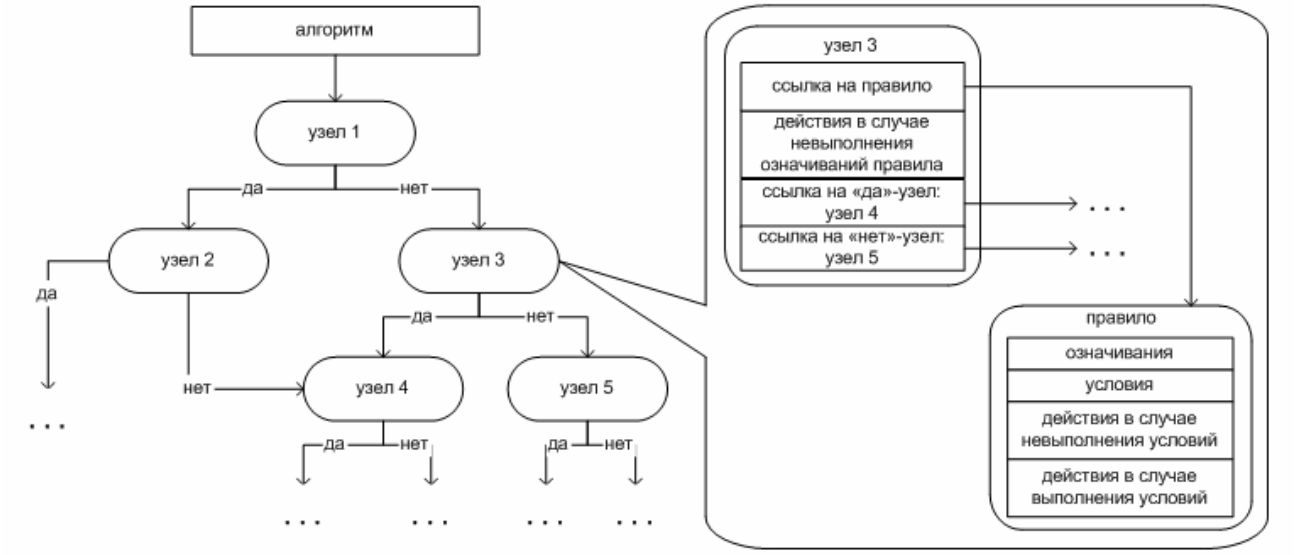

Рис.1. Структура алгоритма и его узлов

В блок-схемах, иногда используемых для спецификации (или для изображения) программ возможны обратные связи, т.е. ссылки на узлы, предшествующие данному. Они появляются в случае изображения циклов и переходов на метку в операторе *goto*. В рассматриваемой модели нам удается обойтись без явного отображения обратных связей благодаря применяемой модели вычислений. В модели используется два вида присваиваний. Первое (*assign*) – это аналог обычного присваивания в программировании, когда некоторая переменная означивается значением некоторого выражения. Второе (*enumerate*) перечисляет все объекты, представляющие члены предложения, удовлетворяющие заданной конъюнкции условий на них. Каждый из этих объектов поочередно становится значением указанной в *enumerate* переменной, причем после каждого такого означивания запускается дальнейшее вычисление подсети алгоритма, начинающейся с узла-владельца данной инструкции *enumerate*. Кроме того, имеется несколько видов инструкций-прерываний таких «перечислительных» вычислений. Окончание работы алгоритма тоже отмечается особым прерыванием. Прерывание проставляется в некоторых узлах алгоритма, заменяя собой ссылки на дочерние узлы. При запуске очередного вычисления из инструкции *enumerate* оно выполняется до ближайшего прерывания. Причем в зависимости от вида прерывания и состояния объектов-перечислителей происходит либо а) возврат к ближайшей инструкции *enumerate*, которая может присвоить следующее значение своей переменной (на пути от узла с перечислителем до узла с прерыванием могут встретиться и другие перечислители – так называемое вложение их друг в друга), либо б) переход к следующему узлу алгоритма, если все предшествующие объекты-перечислители себя «исчерпали», либо в) остановка работы алгоритма и выход из него. По существу, такое перечисление посредством *enumerate* является аналогом инструкции *while* в языках программирования. То, что перечисления применяются здесь только к

объектам, представляющим слова анализируемого предложения, объясняется достаточностью такого выразительного средства для алгоритмов ПСА и позволяет иметь более простую модель вычисления. При необходимости возможно расширение перечислителей и на переменные других типов данных. (Например, для отладки ядра системы используются перечислители натуральных чисел, удовлетворяющих некоторым условиям. Т.е. перечислители реализованы в системе универсальным образом.) Как известно, циклом *while* всегда можно элиминировать цикл, заданный переходом на метку из одного или нескольких операторов *goto*. Т.о., обратные связи на диаграммах алгоритмов Т.Ю. Кобзаревой удается представить при помощи конструкции *enumerate*, а сами алгоритмы представлять сетями.

Реализация объектов модели и связей между ними выполнена на языке Common Lisp с использованием встроенной в него библиотеки CLOS (Common Lisp Object System) в инструментальной среде разработки Corman Common Lisp ® версии 2.5. В языке Лисп имеется возможность в процессе выполнения функций вычислять выражения, записанные в его же синтаксисе. Т.е. некоторое выражение, представляющее собой композицию вызовов Лисп-функций, может какое-то время существовать как данное и даже изменяться или вообще строиться в процессе работы программы, а потом в нужный момент может быть вычислено. Таким образом можно строить код новых Лисп-функций во время работы программы и, при желании, вычислять их при каких-либо значениях аргументов. Это делает синтаксис Лиспа удобным и для записи присваиваний, условий и действий в правилах. Кроме того, «входной» язык для задания алгоритмов и объектного представления анализируемого предложения с учетом альтернатив в смысле ПСА [4] также использует выражения Лиспа.

Например, фрагмент узла с правилом «если слово p, находящееся непосредственно слева от *рассматриваемого* слова W, является предлогом, зафиксировать *рассматриваемое* слово W как существительное» (ясно, что данное правило справедливо лишь в контексте других правил; W присваивается значение в одном из предыдущих узлов алгоритма, оно представляет собой омоним «существительное-глагол») в синтаксисе «входного» языка выглядит так:

```
(:algnode
 :nodeID "node-prep&hom-check"
 :rule
 (:rule
  :ruleID "prep&hom"
  :ruleAssignments
  ((:assign p (:neighbour W :toLeft 1)))
  :ruleConditions
  ((= (:IGO-part-Of-Speech p) 22))
  :ruleYesActions
  ((:fixAs W :Hyp-IGO-part-Of-Speech
             '(1 2)))
 )
 :node-when-ass-failed :repeat
 :yesNode . . .
 :noNode . . .
)
```

Здесь с двоеточия начинаются определенные в системе объектного моделирования ключевые слова, играющие различные описательные роли. Так, `:algnode` и `:rule` выделяют формы, описывающие соответственно узел и ассоциируемое с ним правило. При помощи ключевых слов определяется данные объектов в соответствии с их структурой в формате (<имя поля> <значение>): в узле `:nodeID` – имя узла, `:rule` – правило, `:yesNode` и `:noNode` – ссылки на узлы, к которым следует перейти соответственно при выполнении и невыполнении правила (не показаны в данном примере), `:node-when-ass-failed` – инструкция, обозначающая действие в случае невозможности выполнить присваивания значений каким-либо переменным в правиле (в данном случае `:repeat` требует возврата к вычислению ближайшей в пути от корня к данному узлу инструкции *enumerate*); в правиле: `:ruleID` – его имя, `:ruleAssignments` – присваивания значений переменным, `:ruleConditions` – проверяемые в правиле условия, `:ruleYesActions` и `:ruleNoActions` – действия в случае соответственно выполнения и невыполнения условий правила. Наряду с именами встроенных функций Лиспа ключевые слова используются также для задания системно-интерпретируемых форм в выражениях присваиваний, условий и действий правил. При вычислении эти формы раскрываются в Лисп-выражения (как правило, более громоздкие) с использованием системно-определенных функций, обеспечивающих доступ к представлению анализируемого предложения в модели. В трех формах приведенного примера переменной `p` присваивается указатель на соседнее с `W` слово слева; в условии правила проверяется, является ли `p` предлогом; и в случае выполнения условия в списке свойств объекта `W` свойство `:Hyp-IGO-part-Of-Speech` (список обозначений возможных частей речи омонима) заменятся на свойство `:IGO-part-Of-Speech` со значением из списка `(1 2)`, что обозначает «часть речи существительное» (выбирается то обозначение существительного, которое присутствовало в списке для `:Hyp-IGO-part-Of-Speech`).

Отладка алгоритмов поверхностно-синтаксического анализа

Организация информации в виде набора алгоритмов удобна для того, чтобы сделать подсистему отладки аналогично режиму отладки (Debug) в инструментальных средах разработки программного обеспечения. Как и в последних, основной сценарий отладки здесь реализован как пошаговое выполнение, в нашем случае – алгоритмов поверхностно-синтаксического анализа.

В рассматриваемой инструментальной системе предметом тестирования (не только с целью отладки, а иногда и для проведения эксперимента) являются:

(1) порядок выполнения алгоритмов в общей схеме ПСА,
(2) различные варианты объединения правил в алгоритмы,
(3) отладка алгоритма в смысле организации (взаимосвязи) составляющих его правил,
(4) отладка отдельных правил (их предметного содержания) в контексте использующих их алгоритмов,
(5) отладка правильности Лисп-кодов, реализующих присваивания, условия и действия в правилах.

Перечисленные задачи обеспечиваются следующей функциональностью.

- **Установка прерывания.** Пользователь может прервать выполнение алгоритма, чтобы проанализировать в данной точке его вычисления состояние контекста его выполнения (текущие значения переменных и атрибутов объектов представления анализируемого предложения). Имеется возможность установить прерывание (breakpoint):
  - на входе и выходе либо алгоритма в целом, либо некоторого его узла,
  - после каждого элемента (означивания, условия, действия) правила.
- **Автоматическое выполнение части алгоритма.** При отладке алгоритма естественным образом возникают части, которые не нужно проходить по шагам из-за гарантированного результата их вычисления, однако без них отладить весь алгоритм невозможно. Это могут быть уже отлаженные его фрагменты, его подсеть от начала до узла, при обработке которого происходит ошибка, и т.п. В описываемой реализации можно сразу вычислить ограниченную часть алгоритма, а далее исполнять его пошагово. Для этого в нужных местах устанавливаются прерывания. При запуске алгоритм автоматически выполняется, пока не встретится прерывание (а при его отсутствии – до конца). Аналогично после продолжения вычисления алгоритма из некоторой точки он также вычисляется до следующего прерывания, либо до конца. Заметим, что здесь останов может произойти не только по назначенному пользователем прерыванию, но и вследствие исключительной ситуации в Лисп-коде элементов правил (ошибки или установленного программистом прерывания).
- **Пошаговое выполнение алгоритмов.** В этом режиме по командам пользователя можно либо вычислить узел алгоритма целиком и в зависимости от выполнения условия ассоциированного с узлом правила перейти к следующему узлу, либо «войти внутрь» узла и пошагово выполнять элементы связанного с ним правила.
- **Просмотр контекста выполнения.** При пошаговом выполнении алгоритма на каждом шаге имеется возможность просмотра:
  - значений переменных,
  - содержания узла и связанного с узлом правила,
  - текущего состояния представления анализируемого предложения.
- **Пробное вычисление выражений элементов правил.** Как было видно на примере приведенного выше узла, условия и действия правил могут быть довольно сложными (форма `:fixAs`). Т.е. сложным может быть или само Лисп-выражение, или фукциональная семантика используемых системно-интерпретируемых форм. В частности, это бывает следствием того, что при вычислении последних выполняются преобразования модели анализируемого предложения. Для отладки таких выражений актуальна возможность попробовать вычислить их или некоторые их подвыражения отдельно от вычисления всего алгоритма так, чтобы не изменить при этом саму модель предложения. Следует отметить, что такие системно-интерпретируемые формы могут дописываться прикладным программистом в процессе развития корпуса алгоритмов. Иногда они могут заменять собой (для более эффективной реализации) целые алгоритмы или некоторые их часто используемые фрагменты, если разработчики уверены в их отлаженности и неизменности в дальнейшем. Кроме того, если нечто, что необходимо для написания правил, отсутствует, то таким образом можно пополнить библиотеку используемых функций, расширяя возможности системы и, тем самым, обеспечивая ее открытость.
  Однако такое расширение часто приводит к необходимости решать в контексте алгоритма задачу тестирования (5), для которой особенно актуальна рассматриваемая в этом пункте опция.
- **Сохранение промежуточных состояний модели анализируемого предложения.** Для того чтобы осуществлять описанную выше возможность пробного вычисления выражений,

модифицирующих представление анализируемого предложения, а затем иметь возможность «как ни в чем не бывало» пошагово или автоматически продолжать выполнение алгоритма с этой точки, необходимо вычислять такие выражения не над самой моделью предложения, а над ее копией. Такая копия, отражающая текущее состояние представления предложения в некоторой точке вычисления алгоритма, может быть сохранена как значение некоторого символа. Этот символ – по существу, глобальная переменная, должен отличаться от символов, участвующих в описании алгоритма. Сохранив такую копию, пользователь может после прекращения работы алгоритма использовать ее для выявления ошибочных ситуаций, для отладки системно-интерпретируемых форм вне алгоритма (что сокращает время отладки, т.к. не нужно каждый раз «подбираться» к месту применения формы), и т.п.

- **Трассировка выполнения алгоритма.** В настоящей версии имеется возможность только получить и сохранить протокол прогона или пошагового выполнения алгоритмов. В будущем предполагается настраивать трассировку с целью подавления ненужных деталей.

Организация процесса отладки

Возникает вопрос: кто является пользователем данной инструментальной системы? Не вызывает сомнения, что «лисповская» нотация в выражениях правил и формализация алгоритмов ПСА в используемой объектной модели если и будут понятны лингвисту, то это должен быть «очень-очень подготовленный» лингвист. Вместе с тем, применение для записи правил универсального языка программирования в совокупности с реализуемыми на нем же системными функциями для решения возникающих в ПСА массовых задач, теоретически позволяет выразить в виде правил все, что эффективно вычислимо. Анализ оригинальной записи алгоритмов ПСА, сделанной лингвистом, показал, что для их представления требуется достаточно выразительный, по сути, универсальный (в смысле языков программирования) язык. Разработка и программная имплементация такого языка является существенно более сложной и трудоемкой задачей, чем реализация рассматриваемой здесь инструментальной системы. При этом, скорее всего, пришлось бы ограничить выразительность такого языка.

Авторы придерживаются мнения, что формализация алгоритмов ПСА, необходимая и достаточная для их программной реализации (хотя бы и в специальной среде для экспериментов с ними) – это прерогатива специалиста иной профессии – прикладного программиста. В работе с рассматриваемой системой роли распределяются следующим образом. Лингвист содержательно разрабатывает правила ПСА, отвечающие тем или иным языковым явлениям, первоначально формируя из них алгоритмы. Вместе с программистом они доводят их до готовности к программной реализации, в данном случае – к представлению в объектной модели. При необходимости программист «дописывает» Лисп-функции, определяя новые системно-интерпретируемые формы. Написанные алгоритмы (новые или варианты уже имеющихся) испытывают, возможно, в различных сочетаниях, на некотором множестве примеров, как с целью отладки, так и для улучшения – упрощения структуры и выражений в правилах, расширения диапазона распознаваемых языковых ситуаций и т.п. При отладке программист выявляет неточности или ошибки в процессе вычисления алгоритмов, локализует их. В зависимости от характера ошибок их исправляет либо сам программист, либо вместе с лингвистом. В случае корректного завершения вычисления лингвист анализирует результаты и, при необходимости, содержательно исправляет алгоритмы, а программист корректирует модель. В итоге, если задача тестирования (5) выполняется только программистом, то задачи (1)-(4), как правило, – специалистом предметной области и программистом вместе, а представленная здесь инструментальная среда облегчает их тесное взаимодействие.

Литература

1) Кобзарева Т.Ю., Лахути Д.Г., Ножов И.М. Сегментация русского предложения. // Труды конференции. Седьмая национальная конференция по искусственному интеллекту с международным участием. КИИ' 2000 Москва. Издательство Физико-математической литературы. 2000.с. 879-880.
2) Кобзарева Т.Ю., Лахути Д.Г., Ножов И.М. Модель сегментации русского предложения. // Диалог'2001. Аксаково 2001. т.2 с. 185-194.
3) Кобзарева Т.Ю. Принципы сегментационного анализа русского предложения. // Московский лингвистический журнал., 2004, Т.8, №1, М.: РГГУ, с. 31-80.
4) Баталина А.М., Епифанов М.Е., Ивличева О.О., Кобзарева Т.Ю., Лахути Д.Г. Инструментальная среда для экспериментов с алгоритмами поверхностно-синтаксического анализа. // Компьютерная лингвистика и интеллектуальные технологии: Труды Международной конференции Диалог'2004(«Верхневолжский», 2-7 июня 2004 г.), М.: Наука, с. 32-38
5) Баталина А.М. Объектное моделирование поверхностно-синтаксического анализа. / Девятая национальная конференция по искусственному интеллекту с международным участием КИИ-2004: Труды конференции. Т.2. – М.: Физматлит, 2004, с. 462-471

6) Ивличева О.О., Епифанов М.Е., Лахути Д.Г. Объектная модель многофункциональных словарей, основанная на синтезе лингвистических единиц. // Компьютерная лингвистика и интеллектуальные технологии: Труды Международной конференции Диалог'2003 (Протвино, 11-16 июня 2003 г.), М.: Наука, с. 223-231