МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное образовательное
учреждение высшего образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт математики, механики и компьютерных наук
имени И.И. Воровича

Направление подготовки
010300 — Фундаментальная информатика
и информационные технологии

Головешкин Алексей Валерьевич

IDE с аспектной разметкой кода для работы с YACC-грамматиками

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Научный руководитель:
к.ф.-м.н., доц. Михалкович Станислав Станиславович

Рецензент:
д.ф.-м.н., проф. Пилиди Владимир Ставрович

Ростов-на-Дону
2015

Постановка задачи

Разработать средство для эффективной работы с грамматическими файлами генераторов компиляторов, реализовать концепцию аспектной разметки кода. Создаваемое средство должно обеспечивать:

- разметку грамматики в терминах пользовательской задачи;
- автоматическое поддержание релевантности разметки при редактировании размеченного текста;
- сохранение разметки отдельно от файлов грамматики и загрузку её при открытии соответствующего проекта;
- навигацию между отдельными элементами грамматики, составляющими аспектов и подаспектов;
- выполнение групп запросов для терминальных и нетерминальных символов.

Оглавление

Введение

«Аспект» — термин, позаимствованный из парадигмы аспектно-ориентированного программирования. Он служит для обозначения функциональности, составляющей единую смысловую сущность, но распределённой по нескольким файлам, классам, участкам кода. Данная работа посвящена рассмотрению проблемы аспектов в контексте разработки языков программирования. Создание языка фактически равносильно написанию компилятора для него. Чаще всего грамматика языка описывается в специальном формате, описание передаётся на вход автоматическому генератору компиляторов, он, в свою очередь, генерирует по этому описанию нужный инструмент. На рисунке 1 представлен пример правила, входящего в описание грамматики, распознаваемой генератором компиляторов GPPG (модификацией генератора YACC).

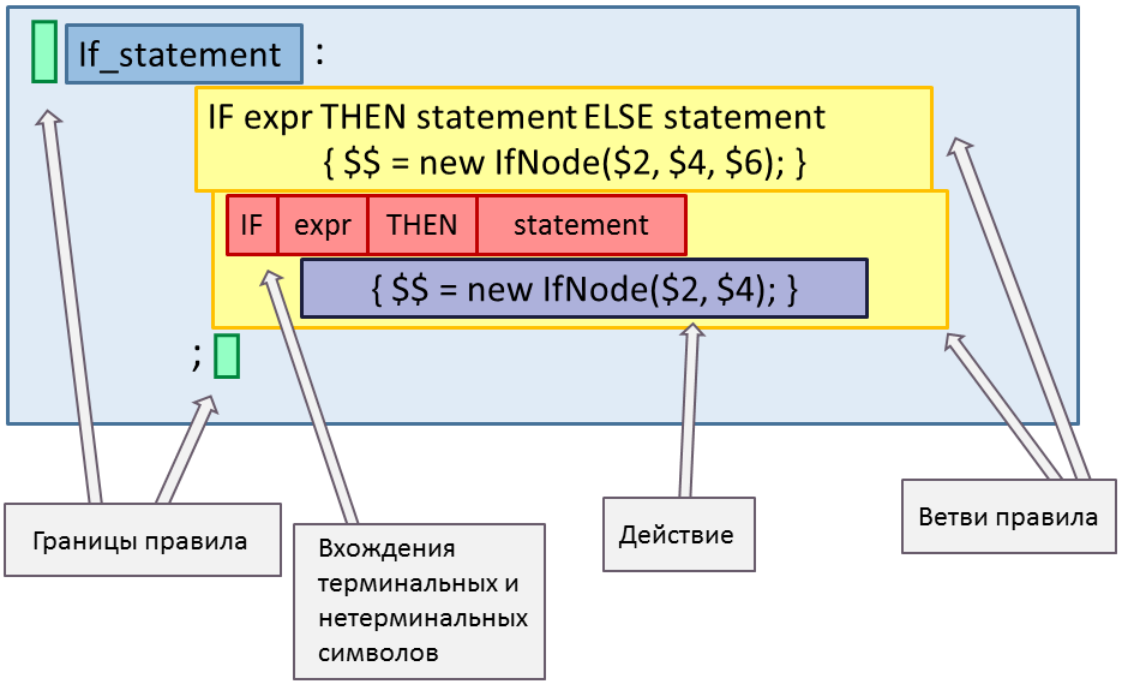

Рисунок 1. Правило грамматики для генератора компиляторов GPPG.

YACC (Yet Another Compiler Compiler) — инструмент, чаще всего используемый совместно с лексическим анализатором Lex. Генерирует код

парсера и лексера, заданных переданной на вход грамматикой, на языке C. Также существует несколько вариантов его модификации: Bison (позволяет получить код на C, C++, Java; использует лексер FLex) и упомянутый ранее GPPG (результирующий код на C#; для лексического разбора применяется анализатор GPLex).

Правило представляет собой форму Бэкуса-Наура, снабжённую дополнительно кодом-действием на том языке программирования, на котором будет сгенерирован компилятор. Код исполняется при сворачивании задаваемой в соответствующей ветке конструкции до стоящего в левой части нетерминального символа, в результате осуществляется синтаксически управляемая трансляция [3, с. 90–99].

Часто аспекты в грамматике возникают естественным образом: для каждого из нетерминальных и терминальных символов важными наборами информации являются данные обо всех вхождениях в грамматику, о зависимых и определяемых сущностях. Более сложные случаи возникают при решении пользовательских задач. Несмотря на простоту отдельных правил, грамматики полноценных языков программирования могут быть достаточно трудными для восприятия ввиду больших объёмов. Так, описание языка PascalABC.NET насчитывает немногим менее 300 правил, в которые в сумме входят около 750 альтернатив. Как показывают исследования [6], в среднем программист тратит от 60 до 90% своего времени на чтение кода и навигацию по нему, при этом более 35% времени занимает переход между участками кода, с которыми ведётся активная работа; чем больше проект, тем больше времени уходит на поиски нужных фрагментов. Данная зависимость может быть перенесена и на процесс разработки грамматик.

При введении в язык очередной синтаксической конструкции требуется добавить к грамматике новые терминалы и нетерминалы, модифицировать уже существующие определения. Однако не всегда ясно, куда

именно в имеющейся линейной структуре описания языка следует поместить новые элементы.

На рисунке 2 представлен набор правил, которые потребовалось внедрить в грамматику PascalABC.NET при добавлении работы с анонимными классами:

```
var p := new class( Name := 'Петров', Age := 2);
```

С одной стороны, логичным является размещение их друг за другом, так как они связаны по смыслу и в тексте должны располагаться рядом. При этом, с другой стороны, разрывается связность уже существующего описания, построенного по этому же принципу. В случае, если программист обладает возможностью оформить каждую новую конструкцию как отдельный аспект, вопрос её размещения в исходном тексте не ставится вовсе, так как все соответствующим образом помеченные составляющие сохранённого аспекта можно быстро найти в любой момент времени.

Существующие классические реализации парадигмы АОП не подходят для решения описанных задач, так как обладают рядом ограничений. Прежде всего, каждая из них ориентирована на конкретный язык программирования, например, расширение AspectJ предназначено для языка Java, PostSharp — для C#. Для языков описания синтаксических и лексических анализаторов подобные расширения отсутствуют. Во-вторых, «расширение» означает не только добавление новой функциональности, но и модификацию синтаксиса: пользователю приходится изучать новые конструкции и ключевые слова, что создаёт дополнительные трудности. Ниже приведён фрагмент аспекта логирования для AspectJ: на три строчки приходится четыре дополнения к базовому синтаксису языка Java.

```
public aspect LogAspect {
    pointcut publicMethodExecuted(): execution(public * *(..));
    after(): publicMethodExecuted()
    …
```

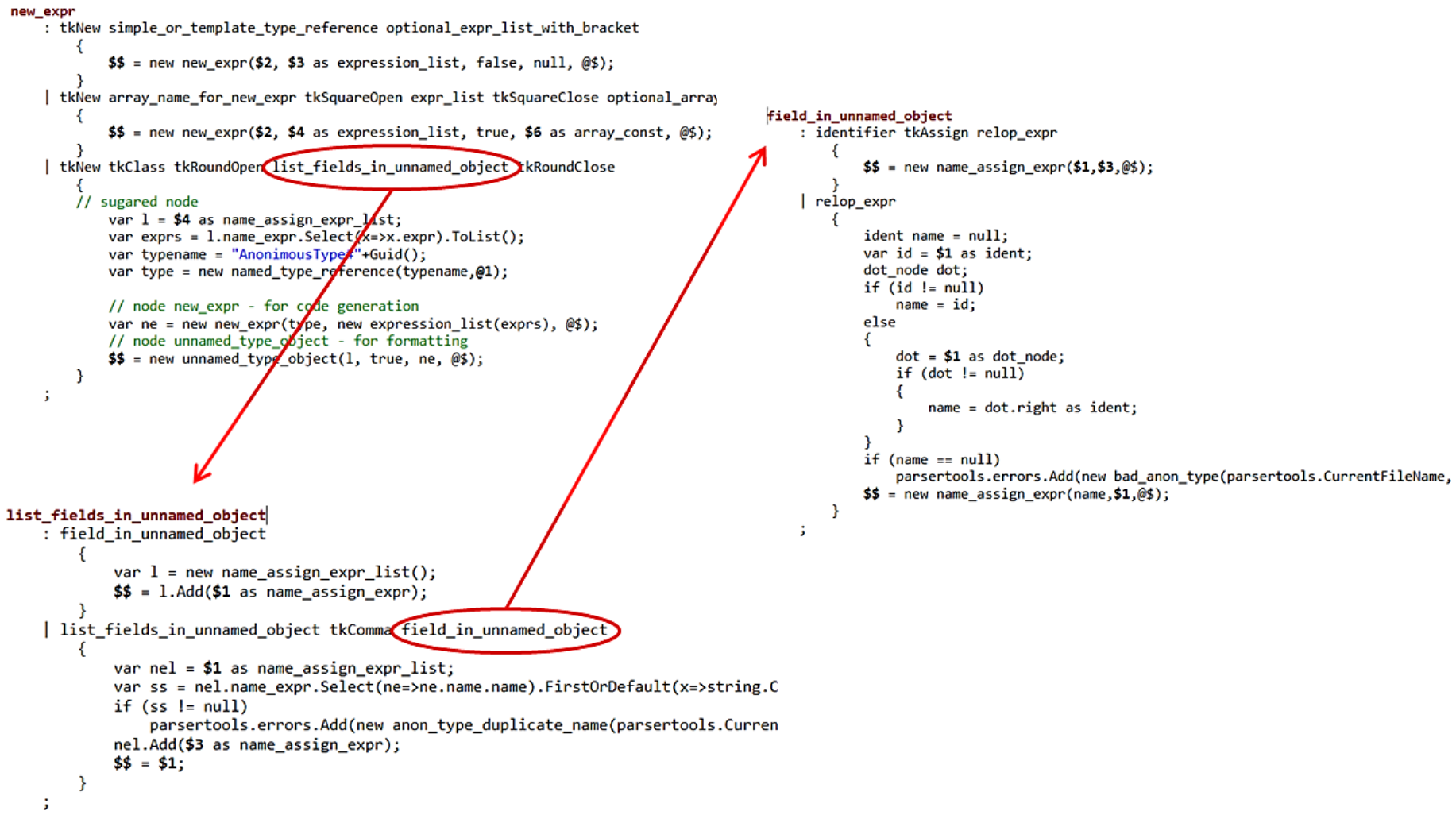

Рисунок 2. Взаимосвязанные правила, необходимые для внедрения одной синтаксической конструкции.

Главным же недостатком в контексте рассматриваемого вопроса является то, что изученные реализации аспектно-ориентированного подхода основаны на идее разделения кода аспекта и кода, отвечающего за основную логику работы [5]. На практике, как уже было продемонстрировано, код аспекта зачастую бывает невозможно выделить из линейной структуры грамматики.

Коллективом разработчиков PascalABC.NET был введён термин «аспектная разметка кода». Это некоторая метаинформация, добавляемая к исходным текстам, но не изменяющая их и хранимая отдельно. Она позволяет группировать логически связанные сущности в аспекты и подаспекты. С пользовательской точки зрения аспектная разметка представляет собой набор интеллектуальных иерархически организованных закладок. Важно отметить, что данная разметка не может быть привязана к текстовым координатам (номер строки и столбца) помечаемых сущностей, поскольку в этом случае любое изменение текста повлечёт за собой потерю ею релевантности. Необходимо учитывать структуру текста, выстраивать по нему некоторые внутренние структуры данных и организовывать по ним интеллектуальный алгоритмический поиск.

Практической реализации изложенной концепции посвящена данная работа. Первая глава даёт представление об интерфейсе создаваемого средства разработки и некоторых теоретических положениях, воплощённых в нём. Во второй главе описан принципиально новый подход к проведению легковесного разбора и генерации легковесных парсеров, опробованный при работе над задачей, а также подробно рассмотрено построение структур данных, используемых для привязки аспектной разметки. Третья глава посвящена различным типам разметки, которые были получены в результате.

Глава 1. Интерфейс

Два года назад автором данной работы была создана среда YACC MC (YACC More Compilers) — оболочка для автоматического генератора компиляторов GPPG, позволяющая осуществлять редактирование файлов грамматики, простейшую навигацию и преобразования. Её интерфейс (рис. 3) был оставлен без изменений и использован при решении новой задачи.

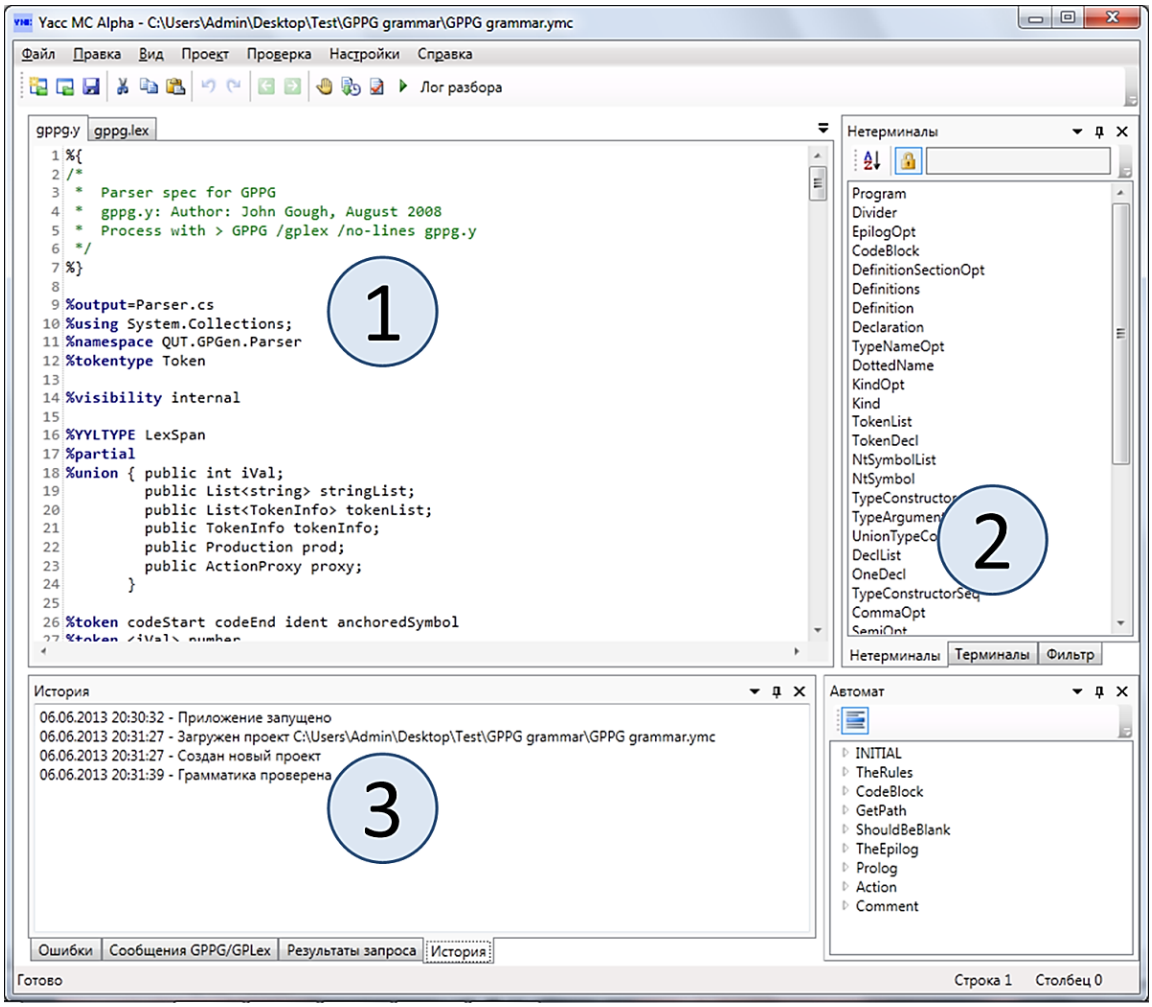

Рисунок 3. Основное окно среды YACC MC: 1 — область редактирования; 2 — информационная область; 3 — область вывода.

Основное окно программы разделено на три части: область редактирования, информационную область, область вывода. Область редактирования предназначена для работы с содержимым файлов грамматики. В качестве компонента, отвечающего за отображение и изменение текста, ис-

пользован редактор с открытым исходным кодом AvalonEdit. Информационная область содержит данные о выделенных в грамматике аспектах. Следует отметить, что множество аспектов не является однородным, различные их категории распределены по нескольким смысловым вкладкам. Между вкладками возможен перенос данных, также для выбранного аспекта и его составляющих — элементов грамматики осуществляются навигация и преобразования. Область вывода отображает информацию об ошибках, обнаруженных при разборе, результаты выполнения пользовательских запросов, сообщения используемого генератора компиляторов, историю работы с программой.

В рамках описания грамматики можно выделить два относительно независимых «горизонтальных» [1] слоя: спецификацию лексического анализатора и спецификацию синтаксического. В рамках каждого из них также выделяются свои слои: секция описаний, секция правил и секция кода. Область редактирования, таким образом, позволяет осуществить доступ к проекту в горизонтальном направлении. Информационная же область обеспечивает «вертикальный» доступ: в рамках аспекта могут быть объединены элементы разных файлов и разных секций, при этом данное объединение смысловое и в большинстве случаев относительно независимое от других. Таким образом, реализуется концепция IDE с поддержкой двумерного представления одномерного текста. Стоит отметить, что измерения не являются равноправными: обозреваемый пользователем текст не генерируется автоматически при выборе слоя. Основным направлением является горизонтальное, поскольку проект всегда представлен в виде двух текстовых файлов: yacc- и lex-описания, — однако присутствует «сосредоточенное описание рассредоточенного вертикального слоя» [1] на аспектных панелях и при выводе результатов запросов.

Глава 2. Внутреннее представление

Ранее отмечалось, что аспектная разметка не может быть напрямую привязана к положению помеченных сущностей в тексте, поскольку при таком подходе любое изменение исходного кода влечёт за собой полную или частичную потерю ею релевантности. Базой для аспектной разметки должны являться своевременно обновляемые внутренние представления, хранящие более устойчивую информацию — информацию о структуре размечаемого текста.

Легковесный разбор грамматики

Для построения внутренних представлений проводится легковесный разбор yacc- и lex-файлов. Отказ от использования существующих полных анализаторов указанных форматов продиктован спецификой основной задачи. Применение промышленного компилятора при разборе пользовательского файла делает невозможным построение внутреннего представления для промежуточной версии редактируемой грамматики. Полный синтаксический анализатор предназначен для перевода кода из одного грамматически правильного представления в другое и рассчитан на работу с корректной грамматикой, в случае с разбором грамматики, находящейся в стадии написания или редактирования, необходимо вычленить ключевые элементы её структуры независимо от степени завершённости. Кроме того, работа полного парсера требует значительных временных затрат, что является недостатком при осуществлении разбора «на лету», непосредственно по ходу написания грамматики программистом. Достоинствами легковесного анализатора являются избирательное распознавание сущностей (в подаваемом на вход описании можем выделить только те элементы языка, которые важны для достижения конкретной цели), скорость разбора, а также механизм восстановления, позволяющий продолжить разбор с некоторого места после обнаружения грубых синтаксических ошибок.

Применение генератора LightParse

Стандартная схема написания компилятора состоит из двух этапов:

- создание описаний лексического и синтаксического анализаторов;
- передача их на вход автоматическому генератору и получение кода лексера и парсера на конкретном языке программирования;
- компиляция кода и получение исполняемого файла.

При этом участие программиста необходимо на самом первом этапе: требуется написать два файла, используя при этом три языка: lex, yacc и (в случае применения генератора GPPG) C#. В процессе выполнения данной работы была задействована принципиально новая схема (рис. 4), включающая применение генератора легковесных парсеров LightParse и одноимённого языка описания [2]. По легковесному lp-формату генерируются привычные yacc- и lex-файлы, по ним, в свою очередь, создаются cs-файлы при помощи генератора YACC, далее компилятором C# генерируется библиотека с легковесным парсером, которая и подключается к проекту.

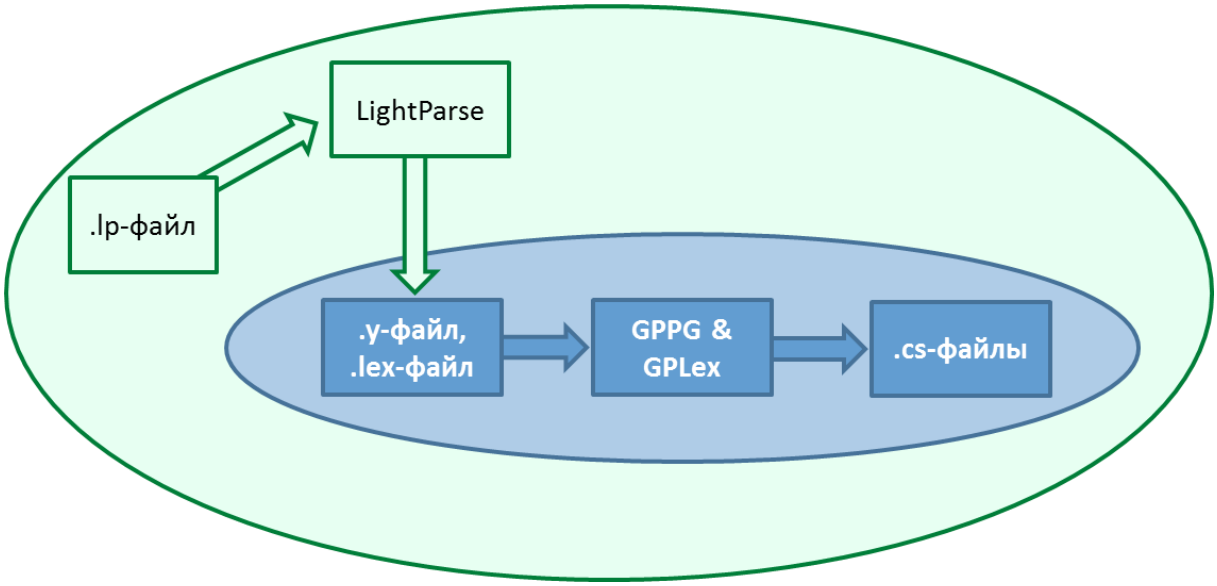

Рисунок 4. Схема генерации легковесного парсера.

В результате вместо двух описаний на трёх языках понадобилось создать всего один файл на языке LightParse, объём необходимого текста

также существенно сократился. В то время как суммарное количество строк yacc- и lex-файлов, описывающих формат yacc, равняется 600, парсер в формате lp насчитывает всего 27 (с учётом пустых строк и комментариев). Для описания lex-формата справедливо примерно то же соотношение.

Спецификация синтаксического анализатора

Полное описание yacc-формата на языке LightParse выглядит следующим образом:

```
%CaseSensitive
%Extension "y"
%Namespace LWYacc

//Пропускаемые сущности
Skip Begin "//"
Skip Begin "/*" End "*/"
Skip BeginEnd "'" EscapeSymbol "'"
Skip BeginEnd '\"' EscapeSymbol '\\'

Token LetterDigit [[:IsLetterOrDigit:]_]+
Token Sign [[:IsPunctuation:][:IsSymbol:]]

Rule Action : @"{" [@Any|@Action]* @"}"
Rule @Program : #Section1 "%%" #Section2 "%%" Any*

//Секция определений
Rule @Section1 : #S1Node*
Rule S1Node :  @"%{" @Any* @"%}"
         | @#"%" #@AnyExcept("%%", "%", "{") #@AnyExcept("%%", "%")*

//Секция правил
Rule @Section2 : #yRule*
Rule yRule :    @LetterDigit ":" #subRule ["|" #subRule]* ";"
Rule subRule :  [@#subRulePart| #Action]*
Rule subRulePart : @AnyExcept("{", "|", ";")
```

В самом начале указывается, что описанный язык является зависимым от регистра, а разбираемые файлы имеют расширение .y, далее задаётся пространство имён, в котором будет находиться генерируемый парсер.

Директива `Skip` служит для «деятельного» пропуска некоторых сущностей, встречаемых в тексте: они распознаются на уровне лексера, но не возвращаются синтаксическому анализатору. В уасс-файле таким образом игнорируются комментарии, строки и литералы. Данный шаг является необходимым, поскольку внутри пропускаемых конструкций могут находиться последовательности символов, совпадающие со значимыми частями грамматики, но ими не являющиеся (например, не нужно распознавать как часть грамматики закомментированное определение или считать концом кода-действия закрывающую фигурную скобку, находящуюся внутри строки). Директива `Token` аналогична описанию лексических категорий в lex-формате.

При помощи директив `Rule` описывается структура анализируемого текста. В частности, в строке

```
Rule @Program : #Section1 "%%" #Section2 "%%" Any*
```

говорится, что спецификация синтаксического анализатора состоит из трёх секций, разделённых символами `%%`, первая и вторая секции затем детализируются далее, третья же — пользовательский код — распознаётся благодаря специальной конструкции `Any`. Наряду с `AnyExcept`, `Any` сигнализирует о том, что в некоторой части текста допустимы любые описанные терминалы, кроме идущих непосредственно за ней или переданных `AnyExcept` в качестве параметров. Таким образом, иногда проще описать разбираемый участок, перечислив не то, что мы ожидаем там увидеть, а то, чего там быть не должно. Примером может служить правило

```
Rule S1Node :  @"%{" @Any* @"%}"
    | @#"%" #@AnyExcept("%%", "%", "{") #@AnyExcept("%%", "%")*
```

В соответствии с документацией [7], элементом секции описаний уасс-файла может быть как код, заключённый в парные кавычки, вида `%{ … %}`, так и опции

```
%имя_опции тело_опции
```

причём после имени в зависимости от него может идти совершенно разное содержимое. В грамматике полного парсера данная проблема решается перечислением всех возможных типов опций (каждая в отдельной ветке правила) и заданием для каждой опции в частности её структуры. При легковесном разборе распознать все элементы секции описаний также важно, но с точки зрения содержимого различать их не имеет смысла. В подробностях нам интересны только несколько видов описаний: `%type`, задающее типы нетерминальных символов, `%token`, служащее для объявления терминалов, `%left`, `%right` и `%nonassoc`, также описывающие терминальные символы и устанавливающие тип ассоциативности.

При этом даже для них не требуется отдельно указывать, что содержимым является список идентификаторов — имён терминалов или нетерминалов. К примеру, для фрагмента вида

```
%using System.Linq;
%namespace GPPGParserScanner
%token <ex> tkInteger tkFloat tkHex
%type <ti> unit_key_word
```

так или иначе будут целиком распознаны все описания, поскольку имеющиеся токены `Sign` и `LetterDigit` покрывают всё множество встречающихся конструкций и не входят в перечень того, что не подбирается при помощи соответствующих `AnyExcept`. В частности, в первой строке символ `%` соответствует описанному в правиле `S1Node` началу `"%"`, затем как `LetterDigit` распознаётся имя опции `using`, что соответствует конструкции `AnyExcept("%%", "%", "{")`. Дальнейшая последовательность лексем

```
LetterDigit Sign LetterDigit Sign
```

удовлетворяет заключительной части `S1Node` вида `AnyExcept("%%", "%")*` — «ноль или более лексем, не являющихся разделителем секций файла или началом новой опции».

По первому подобранному идентификатору для каждой строки можно определить имя описания и, если оно попадает в перечень интересующих нас, дальше перебирать распознанные элементы. Такой перебор начнётся при встрече `token` из третьей строчки. В случае, когда следующий распознанный элемент — символ `<`, мы знаем, что далее будет имя типа, а после — перечень символов грамматики. Если же за именем опции сразу идёт идентификатор, мы встретили просто список терминалов.

Присутствие описания для третьей секции необходимо в связи с тем, что сгенерированный LightParse лексер имеет, с точки зрения описывающего его программиста, только одно состояние конечного автомата. В этом состоянии подбираются все объявленные токены и ключевые конструкции (те, что введены в правилах как строки в кавычках), соответственно, если сформулировать стартовое правило как

```
Rule @Program : #Section1 "%%" #Section2 "%%"
```

при непустой третьей секции произойдёт ошибка разбора, поскольку там гарантированно будут идентификаторы, а значит, несколько раз на уровень синтаксического анализатора неожиданно вернётся токен `LetterDigit`.

Нетерминалы `Any` и `AnyExcept` отчасти уравновешивают данную особенность. С одной стороны, описывая токен, нужно быть готовым к тому, что он встречается во многих местах, а не только в том, ради которого и был введён. С другой стороны, не интересующие нас подряд идущие вхождения терминалов могут быть подобраны одним символом, как результат, не нужно перечислять все возможные варианты их расположения и местонахождения.

Важно отметить, что использование LightParse позволяет разделить анализ структуры грамматики и построение внутреннего представления. Древовидная структура (рис. 5) создаётся автоматически: в процессе описания разбираемого формата при помощи специальных символов `@` и `#` достаточно указать, какие сущности будут задавать имя соответствующего

узла дерева и для каких сущностей нужно сгенерировать дочерние подузлы. К примеру, в определении

```
Rule yRule :    @LetterDigit ":" #subRule ["|" #subRule]* ";"
```

для описания синтаксического анализатора указывается, что правило `yRule` состоит из идентификатора, подбираемого как токен `LetterDigit`, двоеточия, разделяющего левую и правую часть, списка альтернатив `subRule`, разделённых символом `|`, причём в этом списке гарантированно есть хотя бы один элемент.

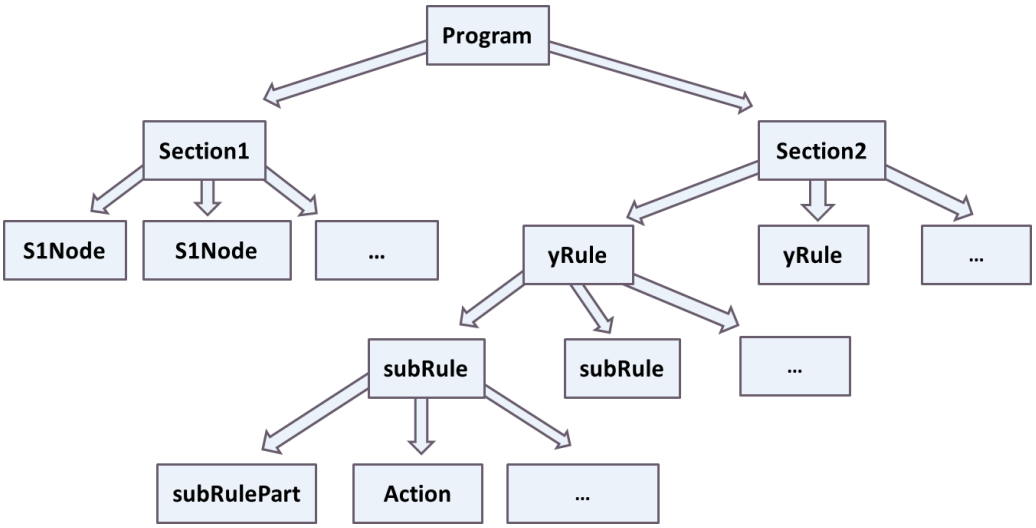

Рисунок 5. Иерархия узлов дерева, генерируемого LightParse для yacc-файла.

Идентификатор `LetterDigit` становится именем узла дерева, соответствующего разобранному правилу, а информация об альтернативах `subRule` помещается в дочерние подузлы.

Спецификация лексического анализатора

Ограничением инструмента LightParse, выявленным при выполнении данной работы, является его сильная зависимость от структурированности текста в терминах ключевых слов и конструкций. Основной сложностью при разборе lex-формата оказывается значимая структурирующая роль отступов, пробельных символов и переходов на новую строку. С одной стороны, их нужно учитывать, поскольку, к примеру, именно переход на новую строку обозначает конец кода в фрагменте

```
"," return (int)Tokens.COMMA;
```

С другой стороны, единожды описанные терминальные сущности, как уже было сказано ранее, подбираются всегда независимо от контекста, поэтому, описав токен

```
Token NL   [\r\n]+
```

который встречается в lex-спецификации столько раз, сколько в ней строчек, мы вынуждены думать о том, в каких смысловых местах во всём описании лексического анализатора он может встретиться, и включать его в соответствующие правила, иначе произойдёт ошибка разбора. Конструкции `Any` и `AnyExcept` упростить ситуацию здесь уже не смогут, так как при их использовании принципиально важно не захватить при подборе начало следующей строки, а значит, придётся описывать внушительные списки исключений.

В результате правила lp-описания приобретают вид:

```
//Lex-файл
Rule @Program: #Section1 tkPC2 #Section2 tkPC2 Any*
...
//Секция определений
Rule @Section1: #Section1Item*
Rule Section1Item: #StateDecl |#RegexDecl |DirectCode |PercentDecl
     |NL |error NL
Rule RegexDecl: @LeftID #Regex1 NL
Rule StateDecl: [@tkPCX | @tkPCS] #IdList NL
Rule PercentDecl: tkPC @AnyExcept(tkBOpen)*     NL
//Секция правил
Rule @Section2: #Section2Item*
Rule Section2Item: #RuleOrGroup | DirectCode | NL | error NL
Rule States: tkLT [@IdList | @Sign] tkGT NL*
Rule RuleOrGroup: @States? NL* [#Group | #LexRule]
Rule LexRule: @#Regex2 NL* ContextCode?
Rule @Group: tkBOpen [#RuleOrGroup | NL | error NL]* tkBClose
```

Ещё одной сложностью является распознавание регулярных выражений — ключевых элементов спецификации лексера. В полной грамматике lex-формата подбор регулярных выражений на этапе лексического анализа осуществляется одним паттерном, который распознаётся только при переходе анализатора в специальное состояние. Ввиду того, что у сгенериро-

ванных LightParse лексеров состояния отсутствуют, одно регулярное выражение, подбирающее все регулярные выражения, поглощает и многие другие сущности, что делает разбор невозможным. Необходимый паттерн был разбит на несколько составляющих, объединённых в набор правил `Rule`:

```
Rule RegexHead: @RegexItem |@Sign |@tkPC |@tkGT |@ID |@LeftID
Rule RegexTail: @RegexHead|@tkPC2 |@tkLT |@tkPCX |@tkPCS
Rule Regex1 : @RegexTail+
Rule Regex2 : @RegexHead @RegexTail*
```

Почти все использованные здесь символы могут быть встречены в грамматике сами по себе, не как части регулярного выражения, и имеют в этом случае самодостаточную смысловую нагрузку. Терминал `RegexItem`, был введён для описания фрагментов паттерна, которые не подбираются имеющимися регулярными выражениями и не могут быть встречены в тексте lex-файла в другом контексте:

```
Token RegexItem \{[0-9]+(,[0-9]*)?\}|\{{ID}\}|\[:{ID}:\]|
\[([^\[\]\r\n]|\[:{ID}:\])+\]|\\[^\r\n]|\"([^\"]|\\\")*\"|"{-
}"|"{+}"|"{*}"
```

Для упрощения на схеме генерируемого по lex-формату дерева (рис. 6) узел-выражение `Regex` показан как одна сущность. На самом же деле, как видно из приведённых фрагментов lp-файла, регулярные выражения в секции описаний и в секции правил подбираются по-разному и представляют собой две разные директивы `Rule: Regex1` для раздела описаний, `Regex2` для второй секции. Для секции правил регулярное выражение начинается с символа `RegexHead`, не включающего, в частности, элементы

```
Rule tkLT: @"<"
Rule tkPC2: @"%%"
```

что позволяет не спутать с описанием паттерна разделитель секций или начало списка состояний `States`.

Данный подход, тем не менее, не устраняет все проблемы: существует две ситуации, когда парсер по-прежнему не в состоянии выделить в тек-

сте lex-спецификации регулярное выражение. В соответствии с документацией [8], правило может иметь не только формат

```
регулярное_выражение {код}
```

но и

```
регулярное_выражение код_до_конца_строки
```

или

```
регулярное_выражение |
```

Пробел после регулярного выражения игнорируется при разборе, в первом случае это не является помехой, т.к. код в операторных скобках подходит под определение

```
Rule @ContextCode: "{" [Any | ContextCode]* "}"
```

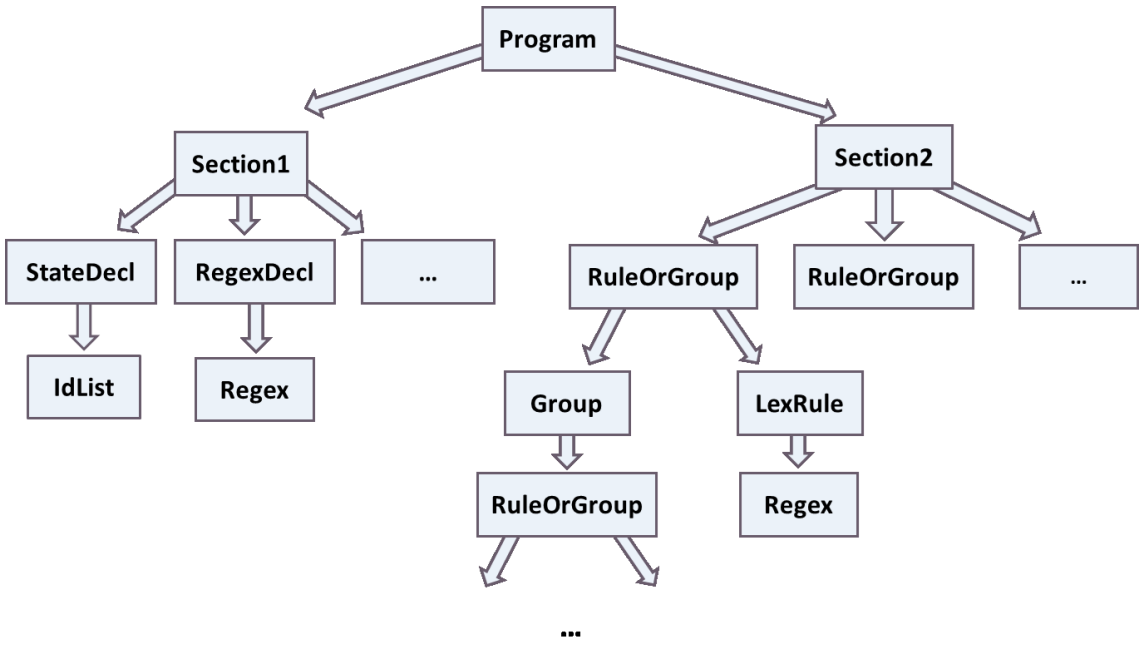

Рисунок 6. Иерархия узлов дерева, генерируемого LightParse для lex-файла.

В последних двух случаях символ | и строка кода подбираются как регулярное выражение, так как подходят под соответствующее правило `Regex2`, как результат, мы получаем «слабое» дерево, в котором узлы `Regex2` иногда охватывают больший участок текста, чем должны. Данную ситуацию приходится исправлять на втором этапе разбора.

Восстановление от ошибок

Одной из причин применения в проекте YACC MC легковесных парсеров стало то, что легковесный разбор менее чувствителен к ошибкам в анализируемом тексте. Говоря о LightParse, стоит отметить, что сгенерированные им анализаторы реализуют несколько стратегий восстановления для случаев, когда ошибки всё же влияют на разбор [2]. Генерируемый LightParse парсер способен добавлять завершающие лексемы и сворачивать частично распознанные правила в случае, если входной поток закончился неожиданно. За счёт этого не происходит потери накопленных за время разбора данных. Второй способ представляет собой делегирование работы по возобновлению разбора стандартному механизму GPPG: места, где можно продолжить разбор в случае возникновения синтаксической ошибки, пользователь обозначает специальным терминальным символом `error`. В случае, когда лексический анализатор возвращает синтаксическому неожиданную лексему, тот переходит в состояние восстановления и поднимается по уровням описанной в lp-файле структуры до тех пор, пока не встретит правило, в одной из альтернатив которого допустим токен `error`. Именно это правило парсер пытается свернуть, основываясь на получаемых далее лексемах (если токен `error` не последний в альтернативе), либо сворачивает сразу и продолжает анализ с первого принятого от лексера подходящего символа. [3, с. 372–374].

При ручном написании файлов для генератора компиляторов можно предусмотреть более тонкие способы восстановления в частных ошибочных случаях. Так, частой ситуацией при написании грамматики является несовпадение числа открывающих и закрывающих операторных скобок в описании действия. Это происходит как вследствие невнимательности программиста, так и ввиду того, что код ещё не дописан до конца. На этапе лексического анализа в этом случае можно подбирать признак начала правила

`^[a-zA-Z0-9_]+[\ \t\r\f\n]*:`

(идентификатор, стоящий в начале строки, за которым идёт двоеточие) при пропуске кода-действия и, если данный паттерн был распознан, считать, что ранее анализируемое правило уже закончилось и далее начинается новое. В данном случае нужно вернуть подобранный паттерном текст во входной поток и сообщить парсеру, что найден конец правила. Это позволит ему свернуть почти разобранную конструкцию и не потерять информацию. Затем данное регулярное выражение будет распознано снова уже в другом состоянии лексического анализатора как штатное начало нового описания нетерминала.

Обработка данной ситуации была успешно реализована и в контексте новой схемы генерации легковесных анализаторов. В lp-описание уасс-файла внесены следующие изменения:

```
Token RuleStart ^[a-zA-Z0-9_]+[\ \t\r\f\n]*:
...
Rule Action : @"{" [@AnyExcept(RuleStart)|@Action]* @"}"
...
Rule yRule : @RuleStart #subRule ["|" #subRule]* ";"
           | error
Rule subRule : [@#subRulePart | #Action]*
```

В правиле `Action` указано, что токен `RuleStart` не может быть частью кода-действия, если же он всё-таки будет возвращён лексером в момент, когда парсер ожидает продолжения описания `Action`, произойдёт ошибка разбора. Восстановиться от неё мы сможем на уровне директивы `yRule`, поскольку здесь в отдельной ветке прописан токен `error`, затем разбор будет продолжен в штатном режиме. К сожалению, при данном подходе теряется информация о правиле уасс-файла, следующем непосредственно за ошибочным, поскольку его начало невозможно отбросить и вернуть во входной поток средствами LightParse. Тем не менее, без описанного механизма неучтёнными оказались бы все нетерминалы, определённые дальше места ошибки.

В случае с парсером lex механизм `error`-ов позволяет восстановиться от «запланированных» ошибок, вызываемых кодом, размещённым в секции описаний и выделенным только отступами от начала строки:

```
...
%option stack
     private int rline;
     private int rcol;
%s codeBlock
IDENTIFIER [a-zA-Z_][a-zA-Z_0-9]*
...
```

Для данного участка каждая из встреченных строчек должна подпадать под действие одного из трёх правил:

```
Rule RegexDecl: @LeftID #Regex1 NL
Rule StateDecl: [@tkPCX | @tkPCS] #IdList NL
Rule PercentDecl: tkPC @AnyExcept(tkBOpen)* NL
```

Начальные токены для них определены следующим образом:

```
Rule tkPCX: @"%x"
Rule tkPCS: @"%s"
Rule tkPC: @"%"
Rule LeftID: ^[a-zA-Z0-9_]+
```

Первая строка распознаётся как `PercentDecl` — определение, начинающееся с символа `%` и идущее до конца строки, в котором не может быть открывающей фигурной скобки. Первая лексема в следующей строке — это идентификатор, однако, в соответствии с описанной грамматикой, идентификатор может быть началом опции только если стоит в самом начале строки (токен `LeftID`). Получив в этом месте обычный токен `ID`, парсер перейдёт в состояние восстановления и, согласно правилу

```
Rule Section1Item: #StateDecl |#RegexDecl |DirectCode |PercentDecl
                |NL |error NL
```

останется в нём до получения токена `NL`, т.е. до момента, когда лексический анализатор начнёт обработку новой строки. В результате в дерево не попадёт лишняя некорректная информация, а парсер сможет завершить начатый разбор.

Второй этап разбора

Узлы дерева, выстраиваемого на первом этапе разбора, представляют собой набор классов: по одному для каждого элемента lp-спецификации, введённого при помощи директивы `Rule`. Классы наследуются от общего предка. Благодаря такой концепции для обхода и анализа удобно применять паттерн Visitor [4, с. 314–328]. Каждый узел включает в себя набор потомков `Items` и коллекцию `Values`, в которую попадают составляющие его имени — имена сущностей, помеченных значком `@` в правиле для соответствующего символа. Также он содержит объект `Location`, представляющий собой структуру с четырьмя полями — координатами начала и конца участка текста, по которому данный узел был сконструирован.

Хотя дерево уже имеет семантику, достаточную для выполнения привязки к элементам грамматики, а именно информацию о структуре и вложенности частей грамматики друг в друга, на следующем этапе по сгенерированной древовидной структуре выстраиваются более специфичные внутренние представления, учитывающие семантику в терминах конкретного формата файла.

Yacc-файл

При помощи визитора `YaccFileBuilder` для дерева, описывающего спецификацию синтаксического анализатора, накапливаются следующие коллекции:

- `Dictionary<string, LexLocation> Rules` — перечень правил грамматики; ключом является имя определяемого нетерминала, значением — текстовые координаты начала и окончания правила;
- `Dictionary<string, LinkedList<SortedList<LexLocation, string>>> Alternatives` — набор альтернатив, описывающих нетерминальные символы; для каждого ключа – имени нетерминала формируется список ветвей правила, ветка представлена

структурой `SortedList<LexLocation, string>` — списком пар «расположение символа грамматики – имя символа грамматики», упорядоченным по положению элементов в тексте.

- `Dictionary<string, LexLocation> Terminals` — словарь координат описанных терминалов;
- `Dictionary<string, IdInfo> Types` — информация о типах терминальных и нетерминальных символов; ключ — имя символа, значение — структура типа `IdInfo`, инкапсулирующая имя типа и место, в котором этот тип ставится в соответствие элементу грамматики.

Находясь внутри узлов дерева, построенного для описания синтаксического анализатора, то или иное действие можно произвести, почти не опираясь на внешний контекст. Для добавления элемента в словарь `Rules` достаточно знать имя нетерминала и координаты, всё это хранит сам узел типа `yRule`; данные о терминалах и типах сосредоточены в узлах `S1Node` — потомках `Section1`. Единственный момент, когда контекст необходим — посещение узла `subRule`: для того, чтобы поставить альтернативу в соответствие некоторому нетерминальному символу, этот символ нужно запомнить при проходе через родительский элемент `yRule`.

Lex-файл

Для описания лексического анализатора ситуация складывается иначе. Как уже было сказано, построенное дерево является «слабым» ввиду недостаточно строгой структурированности lex-файла. В связи с обнаружением данного факта и общей зависимостью генератора легковесных анализаторов от структурированности текста, автором данной работы совместно с автором LightParse был разработан обобщённый механизм, позволяющий в рамках одной «транзакции» — первичного разбора файла — совершить действия по коррекции дерева сразу после его создания. За счёт

этого этап, на котором существует «слабое» дерево, становится невидимым извне легковесного парсера.

Внутри метода `Parse` класса `LightweightParser` (класса сгенерированного легковесного синтаксического анализатора) по завершении разбора вызывается виртуальная функция

```
public virtual void ProcessTree()
```

в её переопределённой версии, если необходимо, должно происходить исправление древовидной структуры. Исходный же метод не содержит никаких действий. В случае, когда требуется совершить коррекцию, к C#-коду, сгенерированному по lp-спецификации, до передачи его компилятору и создания библиотеки прикладывается ещё один файл, содержащий переопределение метода `ProcessTree`:

```
namespace пространство_имён_сгенерированного_парсера
{
    public partial class LightweightParser : LightweightParserBase
    {
        public override void ProcessTree()
        {
           ...
        }
    }
}
```

Одним из наиболее удобных способов исправления неточностей, допущенных при конструировании дерева, является применение специально написанного визитора. В этом случае его определение также должно быть помещено в данный дополнительный файл.

Для lex-формата дерево обходится при помощи визитора `LexTreeChecker`. В ходе посещения узла-правила изменяется его дочерний узел, соответствующий регулярному выражению. Прежде всего, по координатам `Location` в исходном тексте, к которому также имеется доступ, находится строка, разбор которой привёл к появлению данного потомка. Из неё выделяется подстрока с позиции, соответствующей столбцу

начала паттерна. Затем в этой подстроке ищется соответствие регулярному выражению

```
(\\u0022([^\u0022\\]*|(\\\\)+|\\[^\\])*\\u0022|\[([^\]\\]*|(\\\\)+|\
\[^\\])*\]|\\\u0022 \u0022|\\\[|\\\\u0022|\\\\|[^ \n\r\t\f\v])+
```

учитывающему допустимые ситуации, когда пробельный символ может участвовать в регулярном выражении в секции правил lex-файла. Пробел может оказаться частью строки, также он может быть подобран как часть символьного класса `[некоторая_последовательность_символов]`. В рамках этого же паттерна описываются случаи экранирования кавычек и открывающей скобки `[`. В результате для ситуаций

```
регулярное_выражение код_до_конца_строки
```

и

```
регулярное_выражение |
```

подбор остановится на пробеле перед кодом или символом `|`, так будет выделен участок строки, действительно являющийся регулярным выражением. Массив `Values` рассматриваемого узла `Regex2` очищается и вместо набора фрагментов, принадлежащих и не принадлежащих паттерну, туда записывается один элемент — подобранная подстрока. С учётом её длины корректируются координаты `Location`.

К сожалению, правка дерева для lex-формата требует частичного дублирования действий: сначала разбор проходит автоматически, затем некоторые строки приходится выделять и переразбирать вручную. Тем не менее, применение LightParse остаётся более предпочтительным по сравнению со старой схемой генерации легковесных парсеров. Большая часть работы по построению дерева совершается без внешнего вмешательства, кроме того, важным моментом является совместимость получаемой древовидной структуры со средством свободной аспектной разметки, рассмотренным в главе 3.

Далее следует этап конструирования специфичного внутреннего представления. При помощи визитора `LexFileBuilder` собирается следующая информация:

- `Dictionary<string, RegexInfo> Categories` — лексические категории (псевдонимы для часто используемых регулярных выражений), введённые в lex-файле, и их расшифровка — структура `RegexInfo;`
- `Dictionary<string, HashSet<RegexInfo>> States` — список состояний лексического анализатора и множества регулярных выражений, распознаваемых в каждом из них.

Перед началом обхода дерева создаются вспомогательные коллекции:

```
private HashSet<string> DeclXStates;
private HashSet<string> DeclSStates;
private HashSet<RegexInfo> ExprsInAllS;
private Stack<string> CurrentStatesList;
```

Состояния лексического анализатора делятся на две категории: исключающие (объявляются с меткой `%x`) и включающие (`%s`). В состоянии из второй группы подбираются не только помеченные его именем паттерны: распознаются и все регулярные выражения, записанные в lex-файле без какого-либо явного указания списка состояний, в которых они должны подбираться. Для сохранения наборов состояний используются множества `DeclXStates` и `DeclSStates`, упомянутые регулярные выражения запоминаются во множество `ExprsInAllS`. Поскольку список состояний может указываться для правила в отдельности, а также для группы, в которую могут быть вложены другие правила и группы со своими списками состояний, в любой момент времени требуется помнить, в каких состояниях распознаётся текущее регулярное выражение. Для этого поддерживается стек `CurrentStatesList`.

При входе в узел – первую секцию yacc-файла во множество `DeclSStates` добавляются имена `0` и `INITIAL`, они по умолчанию служат для обозначения состояния лексического анализатора, в котором тот находится в начальный момент времени. При посещении узла `StateDecl` по первому элементу его имени (`%s` или `%x`) принимается решение о том, в какое из множеств состояний нужно добавить идущие далее идентификаторы, для каждого из них создаётся `HashSet<RegexInfo>` в словаре `States`. Если же встречен узел, содержащий описание лексической категории, его имя заносится в словарь `Categories` как ключ, а по первому дочернему узлу конструируется описание регулярного выражения. Структура `RegexInfo` имеет интерфейс:

```
public class RegexInfo
{
    public static Dictionary<string, RegexInfo> Categories;
    public static bool DiscloseCategories;

    public RegexInfo(string reg, LexLocation loc);
    public int Line;
    public int Column;
    public Regex RegExpr;
    public string ToString();
}
```

В статическом словаре `Categories`, ссылающемся на одноимённый словарь объекта `LexFile`, данный класс помнит введённые в текущем lex-файле лексические категории. Благодаря этому посредством переопределённого метода `ToString` он может возвращать строковое представление регулярного выражения с произведёнными подстановками паттернов вместо соответствующих имён. К примеру, если в описании лексического анализатора введены категории

```
IDENTIFIER [a-zA-Z_][a-zA-Z_0-9]*
SpaceSymbol [\n\r\t\ \f\v]
```

то выражение

```
^{IDENTIFIER}{SpaceSymbol}*
```

может быть раскрыто как

```
^([a-zA-Z_][a-zA-Z_0-9]*)([\n\r\t\ \f\v])*
```

Необходимость проведения такой подстановки определяется флагом `DiscloseCategories`.

Обработка узла `RuleOrGroup` начинается с проверки его имени — списка состояний лексического анализатора. В них будет распознаваться описанное далее правило или набор правил, расположенных внутри группы. Каждый элемент списка заносится в стек `CurrentStatesList`, в случае, когда вместо конкретных имён обнаруживается символ *, на стек попадают все имеющиеся состояния лексера. Затем происходит обход дочерних узлов; последнее действие в рамках данного метода — снятие со стека помещённых туда элементов. Дочерний узел — это либо группа, в рамках которой снова обходятся узлы `RuleOrGroup`, либо конкретное правило, первым и единственным `Item`-ом которого является регулярное выражение `Regex2`. Составляющие его имени агрегируются посредством LINQ, результирующая строка передаётся для конструирования объекта `RegexInfo`, затем данный объект помещается во множества словаря `States`, соответствующие каждому из состояний на стеке.

По завершении анализа дерева все регулярные выражения, правила для которых описаны вне групп и без указания списков состояний, добавляются ко множествам, связанным с включающими состояниями. Также объединяются в одно множества для состояний `0` и `INITIAL`.

Поиск семантических ошибок

В процессе обхода деревьев осуществляются простейшие проверки семантики. Для кода синтаксического анализатора при посещении очередного узла-правила и узла – описания набора терминалов проверяется ситуация, связанная с повторным определением символа. Также в отдельном множестве сохраняется перечень символов, использованных в правилах грамматики. После обхода дерева из данного перечня исключаются все

имена, сохранённые в `Terminals` и `Rules` (являющиеся описанными в грамматике терминалами и нетерминалами). В случае, если множество остаётся непустым, мы получаем набор нигде не описанных, но задействованных символов и можем сообщить об этом пользователю. Для lex-файла осуществляется контроль использования не объявленных ранее состояний, также при посещении узла-правила проверяется, не было ли уже встречено правило для этого регулярного выражения в том или ином состоянии из стека-контекста.

Глава 3. Виды аспектной разметки

Описанные алгоритмы и внутренние представления позволяют реализовать несколько видов аспектной разметки, каждый из которых имеет свои преимущества и служит для достижения определённых целей.

Автоматическая разметка грамматики

Как уже было сказано, некоторые аспекты задаются самой структурой грамматики, как результат, они могут быть сгенерированы автоматически. Для yacc-файла это, прежде всего, списки терминальных и нетерминальных символов: данные аспекты существуют сразу после второго этапа разбора грамматики в виде структур `Terminals` и `Rules`, их содержимое по умолчанию выводится в информационной области.

Также внутреннее представление поддерживает ряд пользовательских запросов:

```
public class YaccFile
{
    ...
    /* id – имя нужного нам терминала или
    нетерминала */

    //Список вхождений id в правила грамматики
    public List<IdInfo> GetEntries(string id);
    //Символы, от которых зависит id
    public List<IdInfo> GetDependencies(string id);
    //Символы, зависящие от id
    public List<IdInfo> GetDefinitions(string id);
    ...
}
```

На уровень внутреннего представления передаётся имя интересующей пользователя сущности, в ответ возвращается набор точек привязки, соответствующих искомым позициям в тексте. Интерфейс точки привязки выглядит следующим образом:

```
public struct IdInfo
{
    public string Name;
    public int Line;
```

```
    public int Column;
}
```

Структура содержит координаты позиции элемента в тексте: номер строки и столбца, — что никоим образом не нарушает концепцию аспектной алгоритмической разметки, изложенную во введении. В данном случае аспект хранится не как набор статической информации, получаемой по запросу, а как сам LINQ-запрос (рис. 7), то есть способ выделения этой информации из внутреннего представления. После редактирования грамматики и очередного разбора алгоритм вернёт коллекцию точек привязки, релевантных по отношению к новой версии текста (рис. 8), их перечень будет отображён в области вывода основного окна программы.

Для поиска всех вхождений элемента запрос адресуется словарю `Alternatives`, в каждой из веток для каждого нетерминального символа ищется указанное имя, по найденным парам `<LexLocation, string>` создаются точки привязки, которые и возвращаются в качестве результата. Отчасти аналогично происходит поиск зависимых нетерминалов — тех, в чьём определении участвует символ – аргумент запроса. Все альтернативы перебирать не обязательно, достаточно остановиться на первой ветке, содержащей искомый элемент. Результатом является список точек, связанных с началами найденных правил.

Чтобы найти символы, от которых зависит заданный нетерминал `id`, достаточно пройти по списку `Alternatives[id]` и извлечь из коллекций `Rules` и `Terminals` информацию о местах определения встреченных нетерминальных символов и терминалов.

Указанный механизм автоматической генерации аспектов на основе запросов и специфического внутреннего представления позволяет наряду с доступом «на чтение» — анализом имеющегося текста без его изменения — осуществлять элементарный доступ «на запись».

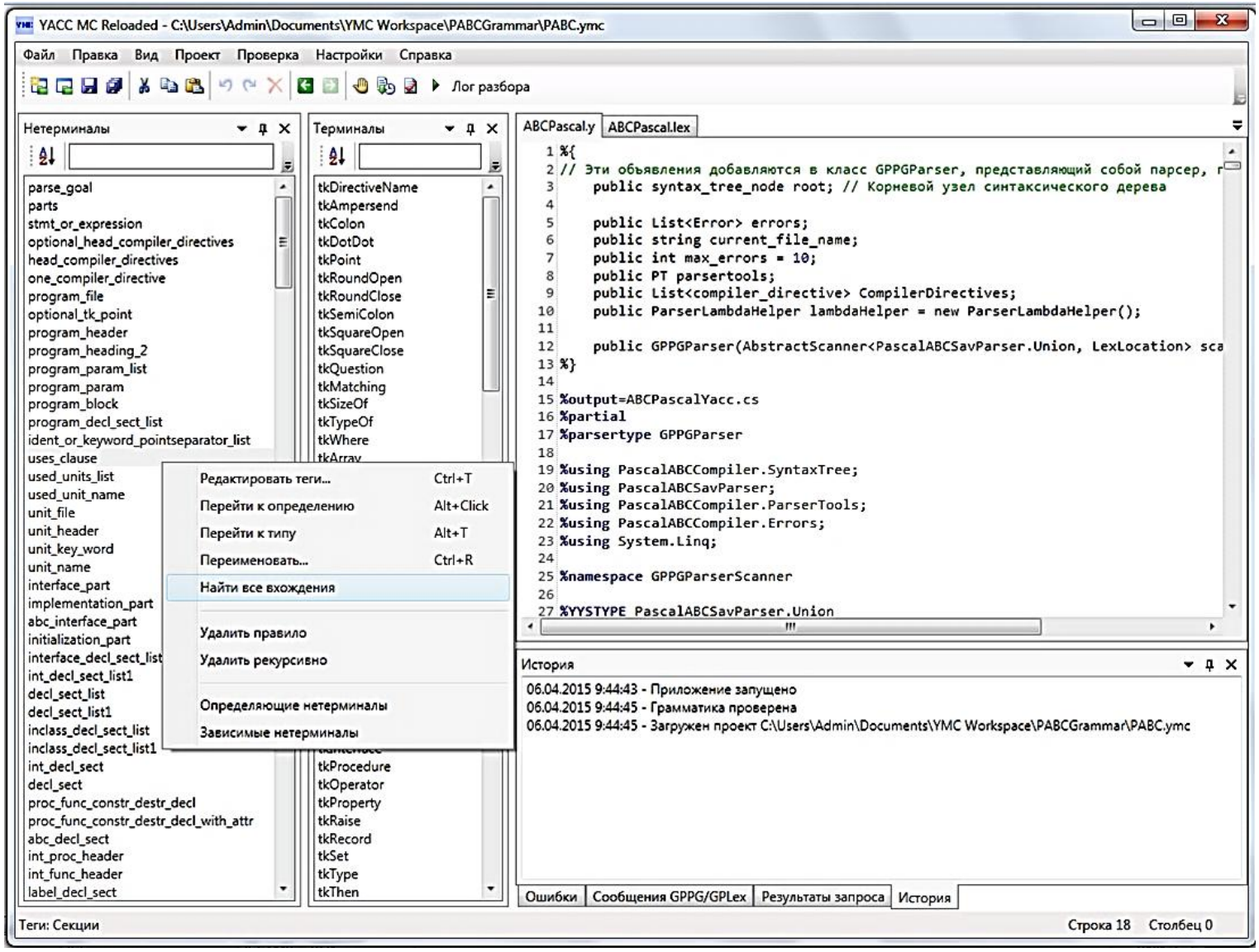

Рисунок 7. Выбор запроса для нетерминала uses_clause.

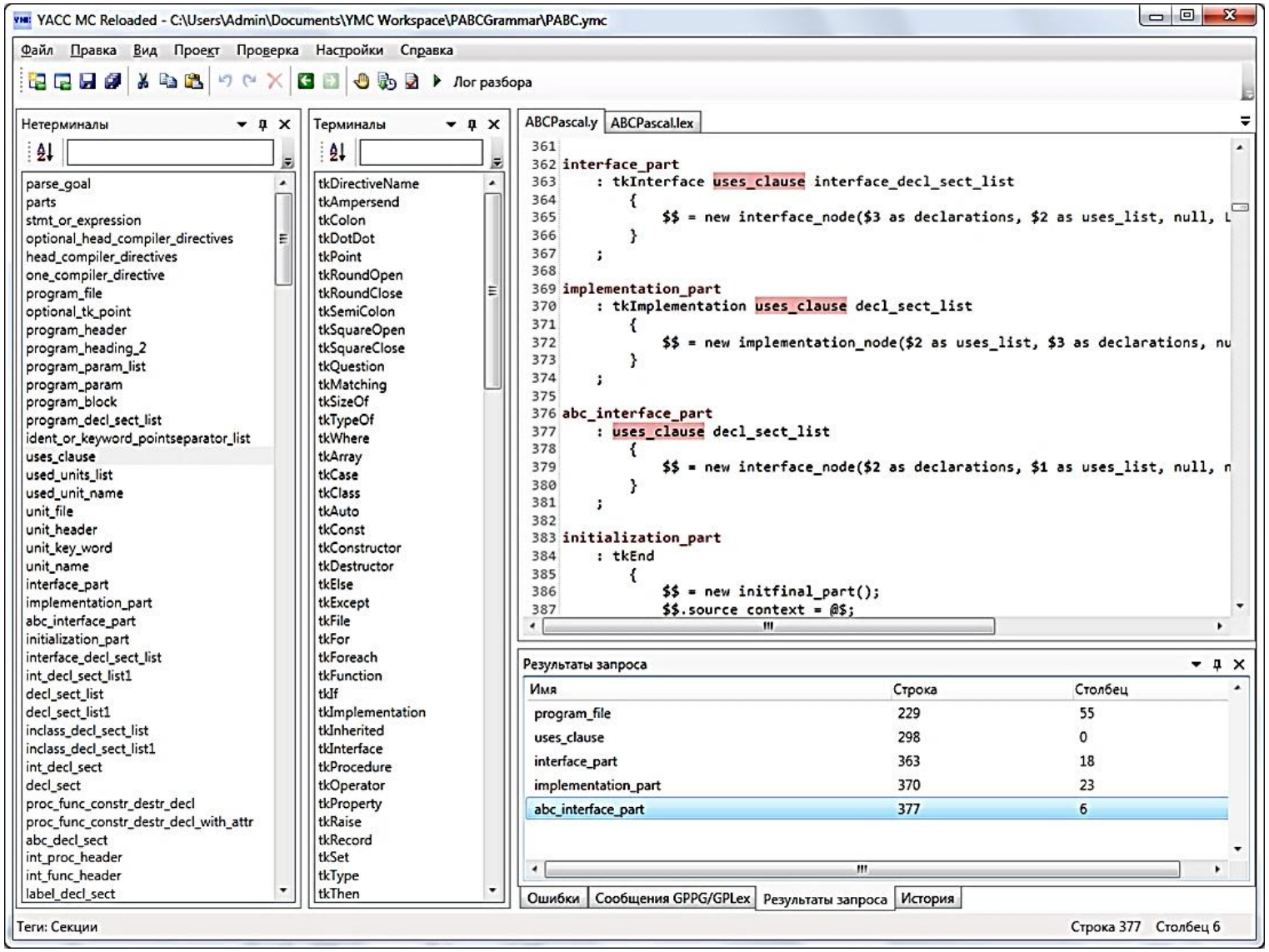

Рисунок 8. Результат поиска всех вхождений нетерминального символа.

Сущности грамматики могут быть переименованы, также аспект может быть полностью выключен — исключён из исходного текста. Рассмотрим особенности данного процесса на примере удаления некоторого экспериментального правила, временно добавленного в грамматику.

При отключении нетерминального символа необходимо, прежде всего, исключить его определение, координаты которого хранятся в словаре `Rules`. Также данный символ может входить в альтернативы других правил, эти альтернативы тоже должны быть удалены. Кроме того, если после удаления подправила правило для некоторого нетерминального символа оказывается пустым, его необходимо изъять из грамматики, таким образом, имеет место каскадное удаление. Технически происходит исключение из текста уасс-файла участков, границы которых содержатся в списке координат, возвращаемом методом

```
public class YaccFile
{
    ...
    public List<LexLocation> DeleteCascade(string id);
    ...
}
```

Данный список формируется в результате анализа внутреннего представления второго этапа разбора. На первом шаге создаётся очередь идентификаторов, впоследствии в неё записываются имена символов, исключаемых из грамматики. На момент создания в очереди оказывается символ, переданный как аргумент запроса; в результирующий список координат включается позиция его определения. Далее, пока очередь не пуста, из неё извлекается очередное имя и осуществляется проход по всем элементам словаря `Alternatives`. Если в какой-то из альтернатив для некоторого нетерминального символа содержится удаляемый элемент, координаты альтернативы попадают в список-результат. В случае, если в списке оказались все ветви правила, определяемый этим правилом нетерминал также заносится в очередь символов, подлежащих удалению, координаты

правила заменяют всю последовательность добавленных к результату координат ветвей.

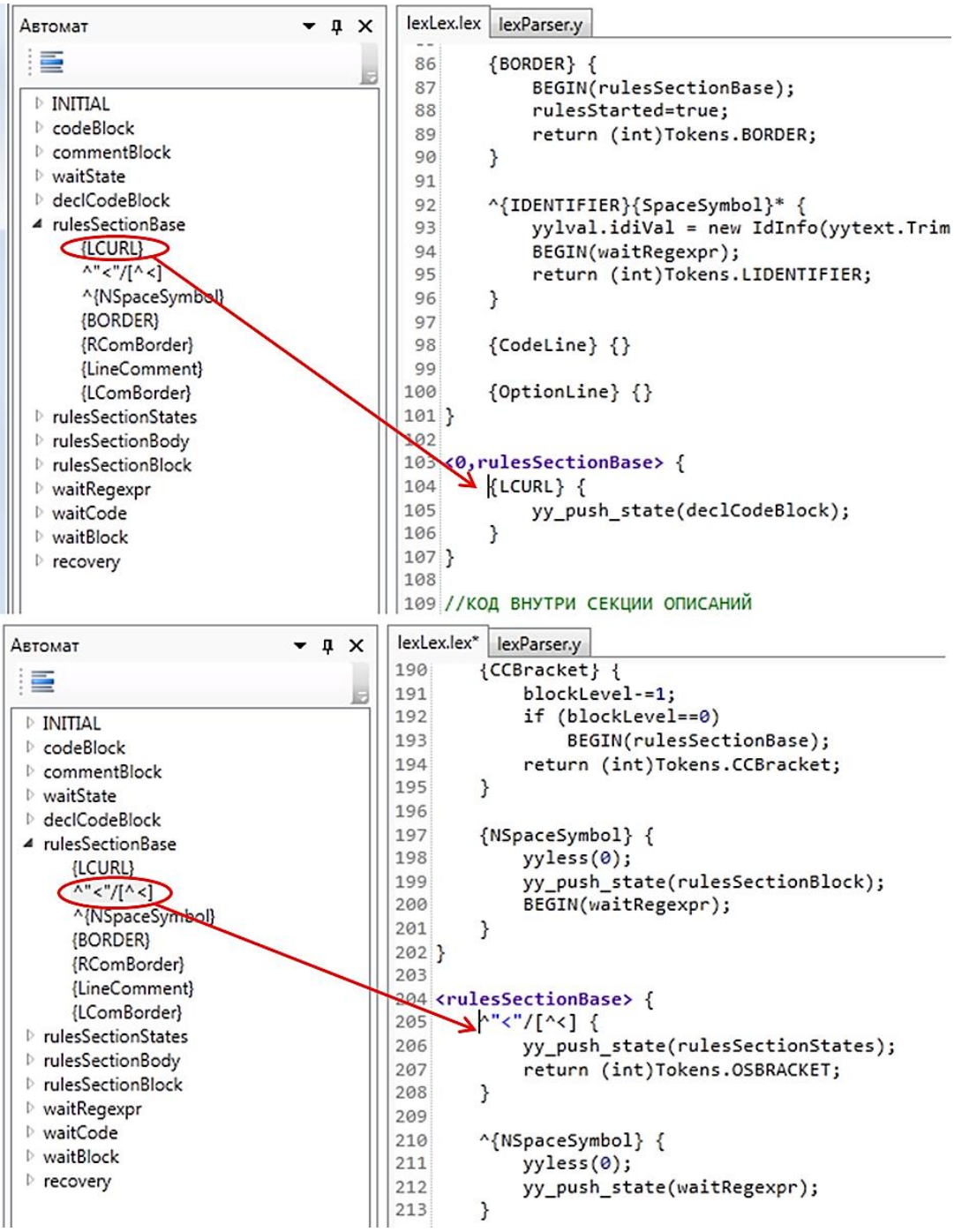

Рисунок 9. Аспект "Регулярные выражения, распознаваемые в состоянии RulesSectionBase".

Для lex-файла автоматические аспекты базируются на словаре `States`. Лексический анализатор, принимая входной поток, распознаёт в

нём указанные программистом регулярные выражения и предпринимает в соответствии с ними некоторые действия, в самом простом случае — возвращает соответствующую лексему парсеру. Множества распознаваемых паттернов могут разниться для различных состояний лексера, причём правила для одного состояния зачастую бывают рассредоточены по тексту грамматики. При помощи специальной аспектной панели (рис. 9), где аспект — каждое состояние лексического анализатора, данная информация собирается воедино. В то время как в самом lex-файле правила могут находиться на достаточно большом расстоянии, таком, что их невозможно увидеть одновременно, перечень соответствующих им регулярных выражений всегда можно найти в одной группе в информационной области.

Панели терминалов, нетерминалов, структуры лексера, а также механизм запросов отчасти реализуют упомянутую в главе 1 концепцию вертикального слоя: они объединяют информацию, собранную при обходе правил, секций файла, при анализе ветвей внутри синтаксического правила. В большей степени этот подход воплощают аспектные разметки, описанные далее.

Теговая разметка грамматики

Преимуществом автоматической разметки является то, что она формируется без участия пользователя и сразу обеспечивает структурирование содержимого грамматики в самом начале работы. Однако данное структурирование происходит в терминах самой грамматики и отношений между её сущностями, зачастую же программисту требуется привнести семантику извне и сформировать аспект для решения собственной практической задачи. Минимальный способ группировки и быстрого поиска некоторого набора сущностей, связанных с решением пользовательской задачи, — добавление к ним тега — строковой метки. Информация о тегах хранится в структуре

```
internal Dictionary<string, HashSet<string>> tags;
```

Ключом является сама метка, значением — множество помеченных терминальных и нетерминальных символов. Ввиду использования класса `HashSet` для представления аспекта, с аспектами уровня теговой разметки можно выполнять основные теоретико-множественные операции (рис. 10).

Множество $\boldsymbol{A}$ — все сущности, связанные с поддержкой константных выражений в PascalABC.NET, множество $\boldsymbol{B}$ связано с константами как таковыми. Если взглянуть на пересечение $\boldsymbol{A} \cap \boldsymbol{B}$, то можно заметить, что почти все константы также помечены и как константные выражения.

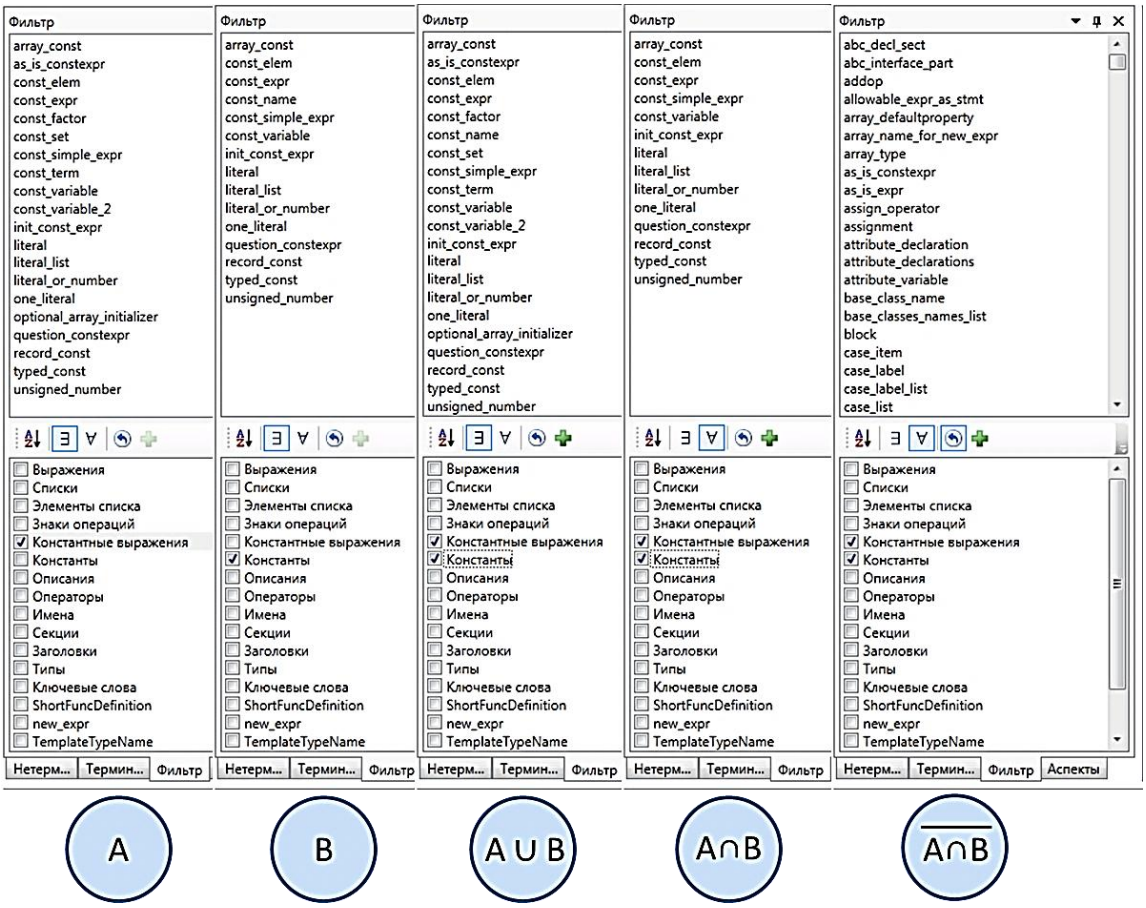

Рисунок 10. Операции с тегами в среде YACC MC.

Добавление тегов к терминалам и нетерминалам позволяет сгруппировать, к примеру, символы, работа с которыми необходима для устранения обнаруженной в грамматике проблемы. На рисунке 11 представлена группа символов, помеченных тегом «проблема с атрибутами», с которой велась реальная практическая работа. Как видно из сопоставления левого и

правого снимков экрана, сущности, составляющие аспект, отстоят друг от друга на расстояние, превышающее 500 строк кода, т.е. в один момент времени программист не может увидеть всю картину, работая только с областью редактирования. Наличие панели фильтрации по тегу в информационной области позволяет всегда видеть список интересующих нас символов и быстро перемещаться между ними.

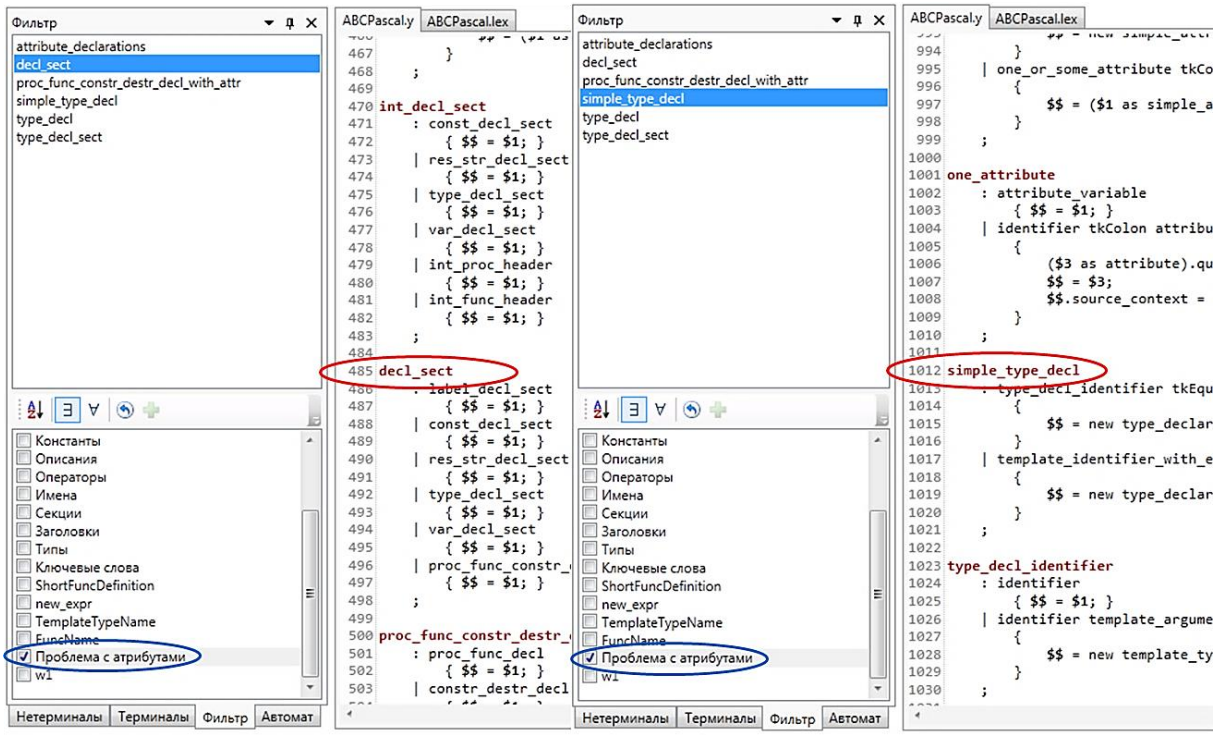

Рисунок 11. Применение тегирования при устранении проблем в грамматике PascalABC.NET.

Данная концепция, по существу, решает более общий вопрос: предположим, в некоторый момент активно ведётся работа над задачей A. Затем программист вынужден переключиться на решение задачи B, к исходной задаче A он возвращается по прошествии значительного промежутка времени. В этой ситуации требуется приложить дополнительные усилия, чтобы вспомнить, с чем именно происходила активная работа в момент переключения с A на B, возможно, данная информация в каком-то виде должна быть сохранена перед самой сменой задачи. Наличие механизма аспектной разметки и теговой разметки в частности позволяет в ходе текущей работы сформировать аспект «рабочее множество» и поместить ту-

да все сущности, которые приходится править и между которыми требуется быстро перемещаться. В случае возникновения новой задачи старый аспект может быть сохранён и впоследствии загружен вновь, что позволяет не тратить время и усилия на восстановление контекста.

Стабильность теговой разметки при изменении файла достигается несколькими способами. Во-первых, при попытке пользователя перейти к символу, помеченному некоторым тегом, запрос адресуется внутреннему представлению, созданному на втором этапе легковесного разбора. Оно своевременно перестраивается при изменениях текста и возвращает актуальные координаты искомой сущности. Во-вторых, в силу того, что словарь `tags` существует отдельно от внутреннего представления, он не затрагивается в случае переразбора файлов. В частности, при удалении терминала или нетерминала из грамматики, символ уже не фигурирует в коллекциях `Rules`, `Terminals` и т.д., однако остаётся во множествах из `tags` до момента, когда пользователь даёт явную команду сохранить текущее состояние проекта. За счёт этого не происходит потери разметки при вырезании участка грамматики и вставке его в другое место либо при редактировании текста с последующей отменой действий.

Сохранение тегов происходит в формат XML, фрагмент файла разметки для грамматики PascalABC.NET представлен далее:

```
<?xml version="1.0"?>
<tags>
      …
      <tag value="TemplateTypeName">
            <name value="type_decl_identifier" />
            <name value="template_identifier_with_equal" />
            <name value="func_name_ident" />
      </tag>
      …
</tags>
```

В отличие от ранее рассмотренной автоматической разметки, которая, по сути, является одноуровневой, разметка тегами имеет два уровня (рис. 12). На первом находятся сами строковые метки, на втором — поме-

ченные ими символы. Очевидно, что один терминальный или нетерминальный символ может быть помечен более чем одним тегом.

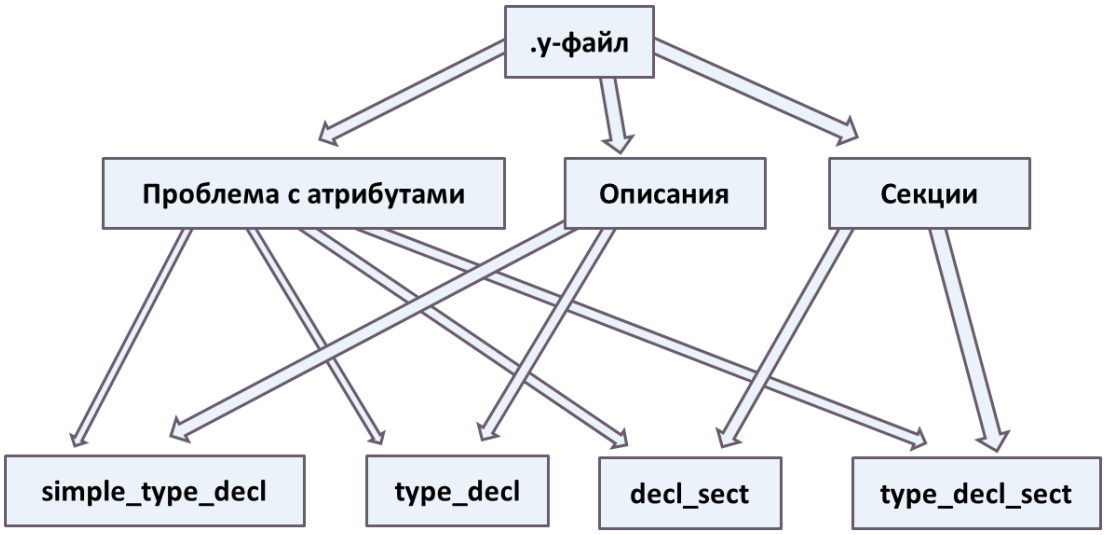

Рисунок 12. Иерархическая структура теговой разметки.

С данным фактом тесно связана проблема удаления аспекта-тега из грамматики. Набор аспектов в данном случае не является однородным [1], поскольку множества элементов, входящих в состав разных аспектов, могут пересекаться. За счёт этого исключение из разметки одного пункта может затронуть и другие. При удалении элемента автоматической разметки существующие взаимосвязи также можно проанализировать автоматически и принять правильное решение, здесь же формальное пересечение аспектов может быть найдено, но только пользователь знает имеющиеся между ними смысловые связи, как следствие, в общем случае указанная проблема неразрешима без получения дополнительной информации.

Свободная разметка грамматики

Любая разметка — это добавление к грамматике некоторой информации, придание частям исходного текста дополнительного значения — добавление семантики. Предыдущие два вида разметки являются довольно простыми, главная причина тому — автоматическое формирование семантики на втором этапе разбора грамматики. Для первого типа разметки хватает семантики, задаваемой форматами yacc- и lex-файлов, второй тип

можно назвать полуавтоматическим, поскольку, хотя пользователь сам определяет необходимую группировку символов, работа при группировке по-прежнему осуществляется с выстроенным на втором этапе разбора специфичным внутренним представлением. Однако зачастую требуется осуществить более тонкую работу с содержимым файлов грамматики. С этой целью в YACC MC была встроена панель [2] для так называемой свободной разметки кода.

Интеграция в среду разработки

Для интеграции панели, помимо установки дистрибутива AspectCore, включающего в себя генератор LightParse, и подключения к проекту соответствующей динамической библиотеки AspectCore.dll, требуется унаследовать класс от `AspectCore.IDEInterop`, имеющего следующий интерфейс:

```
public class IDEInterop
{
    public IDEInterop();

    public virtual string GetCurrentDocumentFileName();
    public virtual string GetCurrentLine();
    public virtual string GetCurrentTextDocument();
    public virtual LexLocation GetCursorPosition();
    public virtual string GetDocument(string FileName);
    public virtual string GetLine(int lineIndex);
    public virtual bool IsDocumentOpen();
    public virtual bool IsDocumentOpen(string FileName);
    public virtual void NavigateToFileAndPosition(string file,
            int line, int col, int lineEnd = 0, int columnEnd = 0);
}
```

Данный класс предоставляет методы, обеспечивающие взаимодействие аспектной панели (визуального компонента, представленного классом `AspectWindowPane`) с текстовым редактором, они должны быть переопределены для каждой IDE, в которую происходит встраивание. Как следует из названий, большая часть методов служит для получения информации об активном документе: его названия, текущих строки и столбца, содержимого строки с заданным номером и т.д. Переход к месту в тексте, со-

ответствующему выбранному аспекту, осуществляется с помощью метода `NavigateToFileAndPosition`.

В целом, интеграция не потребовала значительных усилий, поскольку подходящие архитектурные решения были приняты изначально на этапе проектирования интерфейсной части. В частности, по ходу работы IDE запоминает в отдельной коллекции вкладки, открытые в области редактирования, и в любой момент знает, какой экземпляр редактора является активным. Сама сущность «вкладка редактора» представляется экземплярами класса `DocumentTab`, внутри которых инкапсулируется информация о самом объекте редактора `Editor`, о родительском компоненте `Tab`, на котором он расположен, об открытом файле. Также отслеживаются изменения текста и перемена положения курсора (с тем, чтобы организовать стек перемещений программиста по тексту и возвращаться в место предыдущего редактирования в случае необходимости).

```
public class DocumentTab
{
    ...
    public LayoutDocument Tab;
    public TextEditor Editor;
    public string Filename;

    public bool CanJumpBack;
    public bool CanJumpForward;
    public bool CanCancel;
    public bool WasChanged;
    ...
}
```

Возможности

В отличие от теговой и автоматической разметок свободная аспектная разметка не базируется на обогащённых семантикой внутренних представлениях, полученных на втором этапе разбора. Она использует исходное дерево, хранящее информацию только о структуре файла.

Точку привязки можно создать в любом месте текста грамматики. При этом произойдёт обход дерева и будет найден минимальный по охвату

узел, включающий в себя данный участок. Если программист решил поставить метку внутри правила yacc-файла, эта метка может относиться к нескольким узлам: во-первых, к грамматике в целом, т.е. к узлу `Program` (рис. 5), во-вторых, к секции правил `Section2`, в-третьих, к узлу-правилу типа `yRule` и, наконец, к узлу-альтернативе `subRule`. Все они вложены друг в друга, минимальным по покрытию текста оказывается subRule, к нему и будет предложена привязка.

На рисунке 13 показана схема добавления подаспекта к имеющемуся аспектному дереву (соответствующая панель расположена слева в информационной области). Прежде всего, нужно установить курсор в место, к которому необходимо привязаться, и нажать на кнопку добавления аспекта. В этот момент совершаются описанные ранее действия, в результате в открывшемся диалоговом окне отображается информация о том, к какой строке происходит привязка и какой узел внутренней древовидной структуры соответствует выбранной позиции. Программист может добавить поясняющий комментарий и осмысленное имя для создаваемого элемента, затем узел будет добавлен в аспектное дерево.

Важно отметить принципиальное различие между понятиями «аспектное дерево» и «дерево грамматики»: первое дерево является визуальным и отражает взаимосвязи выделенных программистом точек привязки. Оно обеспечивает переход к ним, учитывает их отношения в терминах вложенности, также заданные автором редактируемой грамматики. Одни точки привязки могут быть добавлены к другим как дочерние либо сгруппированы в каталог, таким образом формируя аспект. Дерево грамматики — ранее рассмотренная невизуальная внутренняя структура, создаваемая автоматически в ходе разбора lex- или yacc-файла.

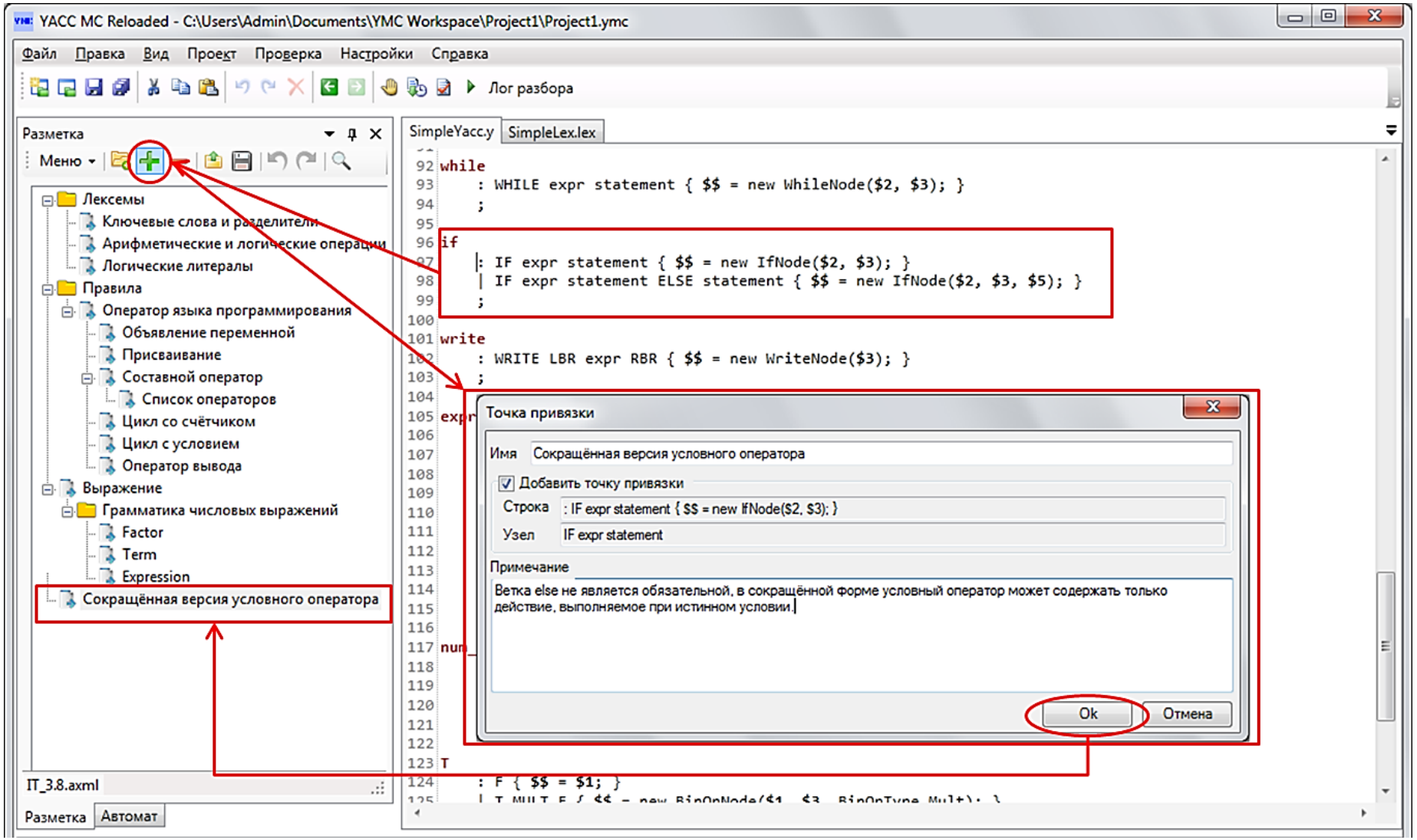

Рисунок 13. Добавление точки привязки в свободной разметке.

В связи со всем изложенным, под сомнение можно поставить одну из ключевых идей, связанных с программированием, — комментирование исходных кодов. По сути, комментарии также можно трактовать как слой разметки, неотделимый от текста, наблюдаемый целиком одномоментно и допускающий только линейное прочтение, а потому в разы менее удобный.

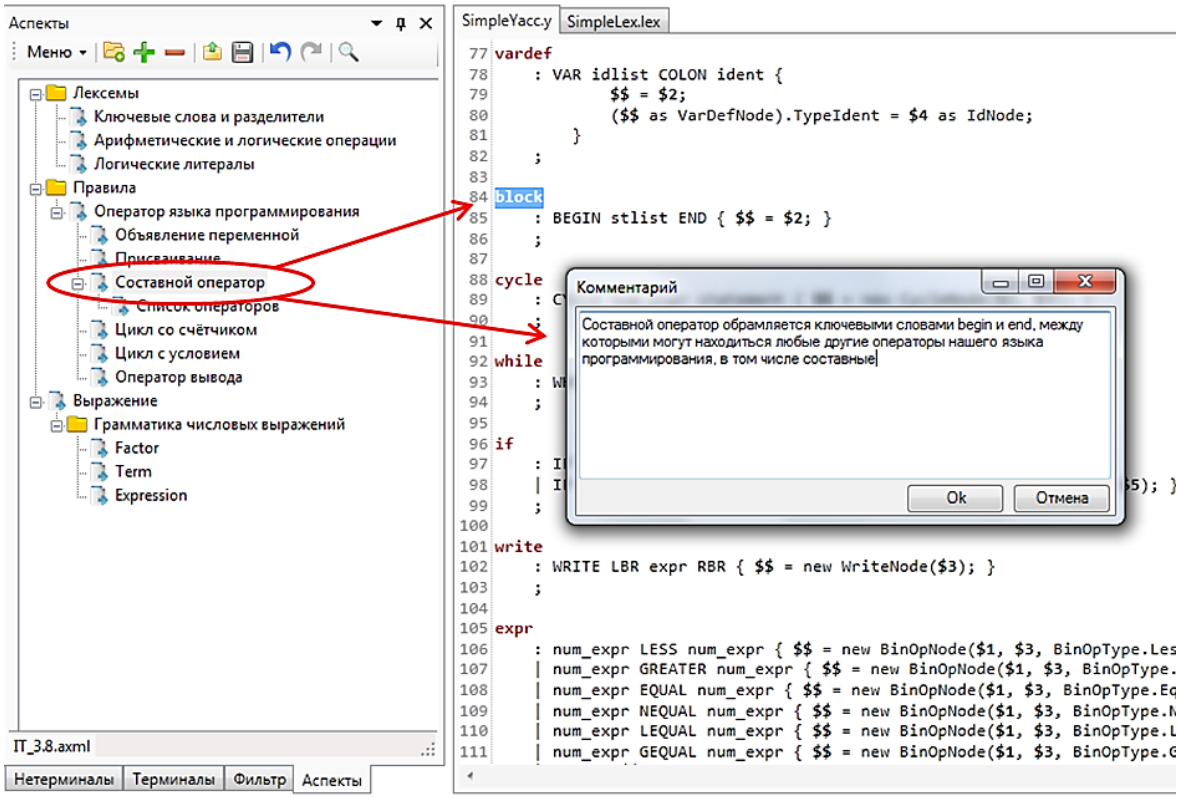

Рисунок 14. Разметка учебной грамматики.

На рисунке 14 показана свободная аспектная разметка учебной грамматики, используемой в рамках курса «Методы разработки оптимизирующих компиляторов». Помечены и прокомментированы основные операторы и ключевые конструкции, при этом комментарий является частью разметки и, как следствие, хранится отдельно от исходных текстов грамматики. За счёт этого, к примеру, имея на руках один экземпляр грамматики, можно разметить его несколькими разными способами и сформулировать необходимое количество индивидуальных заданий по теме.

Фрагмент XML-описания, в которое сохраняется аспектная разметка, представлен далее:

```
<PointOfInterest>
  <FileName>\SimpleYacc.y</FileName>
  <Text>block</Text>
  <Name>Составной оператор</Name>
  <Items>
     <PointOfInterest>
          <FileName>\SimpleYacc.y</FileName>
          <Text>stlist</Text>
          <Name>Список операторов</Name>
          <Items />
          <Note />
          <Context>
               <ArrayOfString>
                    <string>stlist</string>
               </ArrayOfString>
               <ArrayOfString>
                    <string>Section2</string>
               </ArrayOfString>
               <ArrayOfString>
                    <string>Program</string>
               </ArrayOfString>
          </Context>
          <ParserClassName>yRule</ParserClassName>
     </PointOfInterest>
  </Items>
  <Note>Составной оператор обрамляется ключевыми словами begin и
end, между которыми могут находиться любые другие операторы нашего
языка программирования, в том числе составные</Note>
  <Context>
       ...
```

Как можно заметить, элемент `PointOfInterest` — точка привязки — не содержит информацию о том, в какой строке текста и в каком столбце начинается сущность, к которой должен быть обеспечен переход. Вместо этого сохраняется информация о положении сущности в структуре разобранного файла, благодаря чему найти соответствующий ей узел можно и в дереве, построенном по изменившемуся тексту.

В частности, запоминается имя узла (содержимое массива `Values` конкатенируется и помещается в XML-тег `Text`), его тип (тег `ParserClassName`), а также данные о предках (элемент `Context`). Именно вложенность составляющих структуры файла друг в друга является наиболее стабильным фактом, позволяющим отследить нужный элемент даже

при незначительных изменениях его имени либо предложить перечень вариантов для перехода, если исходный узел изменился сильно.

Разные виды аспектов YACC MC не являются изолированными друг от друга, в частности, в аспект свободной разметки может быть экспортирован аспект-тег. API, согласованный с разработчиком комплекса AspectCore, выглядит следующим образом:

```
public interface IAspectWindow
{
    TreeNode AddFolderToTree(string Name, TreeNode Parent);
    TreeNode AddNodeToTree(PointOfInterest Point,
       TreeNode Parent);
    PointOfInterest FindPointByLocation(string FileName,
       int Line, int Column);
    PointOfInterest FindPointByLocation(string Text,
       string FileName, int Line, int Column);
    TreeNode GetSelectedNode();
}
```

Теговая разметка оперирует точками привязки типа `IdInfo`, хранящими информацию об имени помеченного символа и его координатах в тексте. Зная номер строки и столбца, посредством метода `FindPointByLocation` можно получить узел дерева грамматики, соответствующий данной позиции. Затем при помощи `AddNodeToTree` полученную точку привязки, сформулированную уже в терминах свободной разметки, можно добавить к аспектному дереву. В случае, когда параметр `Parent` равен `null`, добавляемый узел становится узлом верхнего уровня, иначе он добавляется потомком к заданному элементу. В частности, дочерний узел можно добавить к активному пункту на аспектной панели, предварительно получив его при помощи `GetSelectedNode`.

В отличие от автоматической и теговой разметки, свободная разметка является древовидной с неограниченным количеством уровней: одни аспекты могут являться подаспектами других, при этом аспекты-каталоги и аспекты-точки абсолютно равноправны с той разницей, что аспект-

каталог сам по себе не предназначен для перехода к позиции в тексте, переходы обеспечивают только его составляющие.

Заключение

В данной работе представлена идея аспектной разметки и её реализация применительно к вопросу разработки грамматик для автоматических генераторов компиляторов семейства YACC.

В процессе выполнения были решены следующие задачи:

- введено понятие «аспектная разметка кода»;
- реализованы легковесные описания yacc- и lex-форматов на языке LightParse, разработаны внутренние представления грамматики;
- на базе внутренних представлений и алгоритмов реализованы три вида аспектной разметки:
    - автоматическая — линейная;
    - полуавтоматическая — двухуровневая;
    - ручная — полностью свободная;
- на основе этих концепций реализована IDE YACC MC для написания грамматик на языках yacc и lex;

Результаты работы были представлены на XXII научной конференции «Современные информационные технологии: тенденции и перспективы развития» [9], а также на студенческой научной конференции «Неделя науки». В настоящее время YACC MC активно используется для работы с грамматикой PascalABC.NET.

Литература

1. Как растёт программа. М.М. Горбунов-Посадов. [Электронный ресурс] // ИПМ им. М.В. Келдыша РАН. URL: http://keldysh.ru/gorbunov/grow.htm (дата обращения: 06.05.2015)
2. Малеванный М.С., Михалкович С.С. Реализация поддержки аспектов программного кода в интергированных средах разработки. [Текст] // Современные информационные технологии: тенденции и перспективы развития: материалы конференции. — Ростов н/Д, 2015. — С. 351–353.
3. Ахо А., Лам М., Сети Р., Ульман Д., Компиляторы: принципы, технологии и инструментарий. [Текст] / А. Ахо [и др.] — М.: ООО «И.Д. Вильямс», 2011. — 1184 с.
4. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж., Приёмы объектно-ориентированного проектирования. Паттерны проектирования. [Текст] / Э. Гамма [и др.] — СПб: Питер, 2001. — 368 с.
5. Малеванный М.С., Михалкович С.С. Поддержка среды программирования для навигации по аспектам программного кода. [Текст] // Научная конференция «Современные информационные технологии: тенденции и перспективы развития». — Ростов н/Д, 2014. — С. 277–278.
6. Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. [Электронный ресурс] // Brown University. URL: http://cs.brown.edu/~spr/codebubbles/icsefinal.pdf (дата обращения: 01.05.2015)
7. The GPPG Parser Generator. [Электронный ресурс] // CodePlex. URL: http://gppg.codeplex.com/downloads/get/378046 (дата обращения: 18.04.2015)
8. Gardens Point LEX. [Электронный ресурс] // CodePlex. URL: https://gplex.codeplex.com/downloads/get/899039 (дата обращения: 18.04.2015)

9. Головешкин А.В., Малеванный М.С., Михалкович С.С. Интегрированная среда разработки грамматики с поддержкой аспектной разметки. [Текст] // Современные информационные технологии: тенденции и перспективы развития: материалы конференции. — Ростов н/Д, 2015. — С. 138–140.