Введение в использование ANTLR

Использование ANTLR для разработки компиляторов и DSL на платформе .Net

Знакомство с ANTLR

ANTLR (от ANother Tool for Language Recognition – Еще один Инструмент для Распознавания Языков) – это инструментарий для создания компилятор или интерпретаторов языков программирования или DSL (Domain Specific Language – язык специфичный для предметной области), инструментов синтаксического анализа (например, статических анализаторов кода) и других языков инструментов.

Создателем и основным идеологом и разработчиком ANTLR, а также ряда связанных инструментов, таких как ANTLRWorks (среда разработки для ANTLR) или StringTemplate (библиотека-шаблонизатор, упрощающая разработку трансляторов DSL, которые генерируют не машинный код, а программу на другом языке), является профессор в области компьютерных наук университета Сан-Франциско Теренс Парр (Terence Parr).

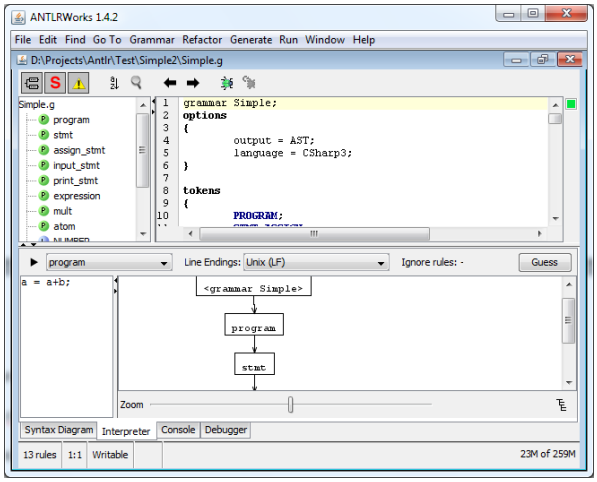
ANTLR является Open Source продуктом и распространяется по лицензии BSD, что позволяет использовать его как в открытых, так и закрытых коммерческих проектах.

Обзор ANTLR

В первом приближении, ANTLR представляет собой набор, состоящий из двух частей:

- **генератора анализаторов** – приложения, которое получает на вход описание грамматики в нотации EBNF (Extended Backus–Naur Form – расширенная форма Бэкуса-Наура) и генерирует код для лексического и синтаксического анализатора.
- **runtime-библиотеки**, которая используется для создания конечной программы. Эта библиотека содержит базовые классы для анализаторов, а также классы, управляющие потоками символов и токенов, обрабатывающие ошибки разбора, генерирующие выходной код на основе шаблонов и многое другое.

ANTLR реализует стратегию нисходящего анализа с использованием ограниченного (LL(k)-анализ) или неограниченного (LL(*)-анализ) предпросмотра входной строки.



Сам генератор ANTLR написан на языке Java, однако он умеет генерировать анализаторы для многих других языков и сред, например C/C++, C#, JavaScript, Perl, Ruby, …. Для всех этих языков (называемых «целевыми» – target languages) имеются портированные версии runtime-библиотеки.

В дополнение к перечисленным инструментом, для ANTLR существует специализированная среда

разработки ANTLTWorks, которая позволяет редактировать (с подсветкой синтаксиса и подстановки кода) и отлаживать грамматики.

Применение ANTLR

Вот несколько наиболее типичных сценариев использования ANTLR:

- разработка статических анализаторов программ (т.е. анализаторов, которые работают с исходным кодом). Это могут быть, например, средства поиска семантических ошибок (например, как VCC) или средства, следящие за соблюдением правил написания кода, принятых в компании (например, как StyleCop)
- разработка модулей разбора (frontend) для компиляторов или интерпретаторов
- разработка трансляторов DSL, которые генерируют код на каком-либо языке высокого уровня
- разработка «трансформаторов» - программ, которые, как и компиляторы разбирают текст на исходном языке программирования, но в результате выдают не машинный код, а тот же исходный текст, но слегка модифицированный (например, переформатированный код или код снабженный заготовками для комментариев, к методам – с уже проанализированными параметрами и возвращаемым результатом)

Полезные ссылки и материалы

Далее приводится небольшая подборка дополнительных материалов, которые могут помочь в изучении ANTLR:

- основной сайт программы http://www.antlr.org/. На нем наиболее полезными могут оказаться следующие разделы:
    - документации – содержит краткое руководство по грамматике, параметрах командной строки, особенностях отдельных runtime-библиотек, и т.д.
    - для скачивания – здесь можно скачать последние версии ANTLR и ANTLRWorks, включая исходные коды.
    - с готовыми примерами грамматик для различных языков. Правда качество разных грамматик также различно.
- книга, посвященная ANTLR: The Definitive ANTLR Reference: Building Domain-Specific Languages by Terence Parr. Это наиболее полное руководство по ANTLR, включающее не только руководство по всему функционалу, но и ряд полнофункциональных примеров. Единственный недостаток, на момент издания книги (2007 год) была выпущена только версия 3.1 и часть функционала, появившаяся позднее, в книгу не вошла.
- краткое пособие по ANTLR, сделанное сторонними авторами.
- сайт библиотеки StringTemplate (http://www.stringtemplate.org/), которая хоть и не является необходимой частью ANTLR, однако тесно с ним интегрирована.

Начало

Прежде чем переходить непосредственно к изучению и работе с ANTLR, мы кратко опишем задачу, на базе которой будем изучать этот инструмент. В качестве таковой мы возьмем создание компилятора для простейшего языка программирования.

Наш язык будет поддерживать три оператора: присваивания, ввода и печати. Пример программы на таком языке объяснит все без комментариев:

```
input a;
x = 3 + a * 5;
print a,  x,  4 + 5 * x;
```

На выходе наш компилятор будет генерировать код на языке IL (Intermediation Language) – промежуточном языке платформы .Net. Например, для приведенной выше программы должен будет сгенерироваться примерно следующий код:

```
.assembly Sample {}
.assembly extern mscorlib {}
.method static void main()
{
        .entrypoint
        .maxstack 100
        .locals init (int32 a, int32 x)

        call string [mscorlib]System.Console::ReadLine()
        call int32 [mscorlib]System.Int32::Parse(string)
        stloc a

        ldc.i4 3
        ldloc a
        ldc.i4 5
        mul
        add
        stloc x

        ldloc a
        call void [mscorlib]System.Console::WriteLine(int32)

        ldloc x
        call void [mscorlib]System.Console::WriteLine(int32)

        ldc.i4 4
        ldc.i4 5
        ldloc x
        mul
        add
        call void [mscorlib]System.Console::WriteLine(int32)

        ret
}
```

Такой подход позволяет довольно наглядно наблюдать за результатом компиляции. Сам наш компилятор будет также предназначен для платформы .Net и написан на языке C#.

Но прежде чем начать работу, необходимо:

- подготовить инструменты и окружение
- познакомиться с основными используемыми в ANTLR

Подготовка к работе

Для работы нам понадобятся:

1. сам ANTLR и среда ANTLRWorks
2. Java-runtime (для работы ANTLR), и опционально JavaSDK (для отладки грамматик)

3. .Net Framework (можно использовать любой, начиная с версии 2.0)
4. runtime-библиотеку ANTLR для .Net
5. Visual Studio (можно версию Express), или любая другая среда разработки под .Net (например, SharpDevelop)

Пункты 2, 3 и 4, скорее всего не вызовут никаких проблем, а вот по пунктам 1 и 4 нужно дать некоторые пояснения.

ANTLR и ANTLRWorks

Сам по себе, ANTLR является консольным Java-приложением, которое принимает на вход файлы с описанием грамматик(и), а на выходе генерирует исходные файлы на указанном языке программирования. В принципе, для работы этого уже достаточно, однако для более удобной работы лучше воспользоваться специализированной средой, такой как ANTLRWorks.

Скачать ANTLRWorks можно с её домашней страницы http://www.antlr.org/works/index.html. ANTLRWorks уже содержит в себе модули ANTLR, а значит, качать его отдельно не нужно.

Если по какой-то причине использовать Java-версию ANTLR нельзя, можно воспользоваться его портом на .Net. Актуальную версию можно найти на странице http://www.antlr.org/wiki/display/ANTLR3/Antlr3CSharpReleases. К сожалению, портирован только сам компилятор ANTLR, но не среда.

Runtime-библиотека

Второй необходимый компонент для работы с ANTLR, это библиотека runtime-поддержки. Получить .Net-версию этой библиотеки можно одним из следующих путей:

- скачать полный порт ANTLR со страницы http://www.antlr.org/wiki/display/ANTLR3/Antlr3CSharpReleases. Я рекомендую брать пакет, который помечен как **antlr-dotnet-tool-<номер версии>.7z** – там сразу есть все библиотеки и сам компилятор ANTLR.
- самостоятельно собрать библиотеку из исходных файлов.

Первый вариант на много проще, однако, бинарные сборки обновляются все же реже и не всегда содержат последние актуальные исправления. Поэтому, иногда бывает выгоднее все-таки собрать runtime самому.

Для этого следует:

1. скачать исходные файлы ANTLR. Это можно сделать, перейдя по ссылке https://github.com/antlr/antlr и нажав кнопку **Download**.
2. в скаченном архиве найти подпапку **runtime\CSharp3\Sources**
3. сгенерировать ключ для подписывания сборок утилитой **sn.exe** (данная утилита входит в состав Windows SDK, который обычно ставится при установке Visual Studio). Сам ключ должен находиться в подпапке **Antlr3.Runtime**
   Для генерации можно использовать примерно такую команду (поправить пути):

   ```
   "C:\Program Files\Microsoft SDKs\Windows\v7.0A\bin\sn.exe" -k Antlr3.Runtime\key.snk
   ```

4. собрать бинарную версию либо через Visual Studio, либо из командной строки. Например, так:

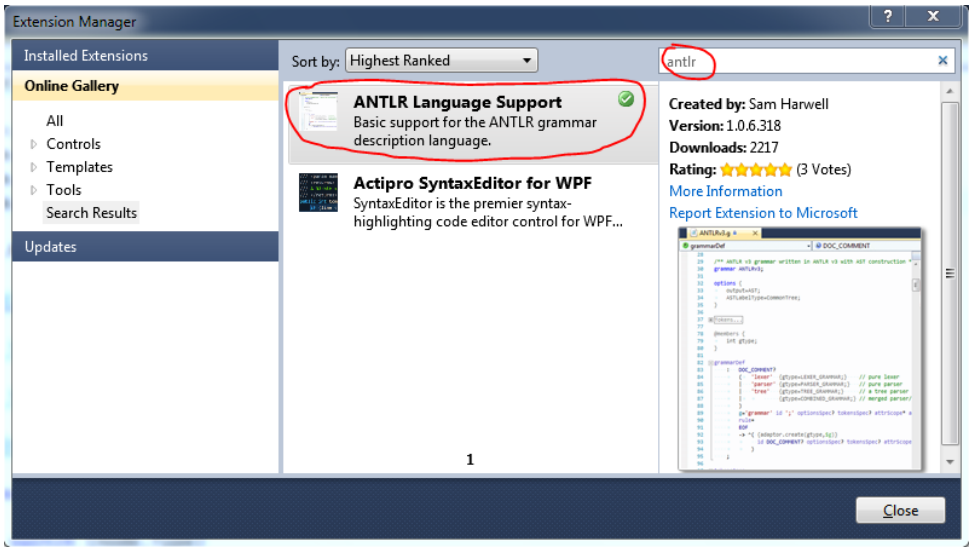
```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe Antlr3.Runtime.sln
```

По большому счету теперь все готово к работе, однако если вы хотите большую часть работы вести в Visual Studio, рекомендую скачать и установить компоненты интеграции с Visual Studio

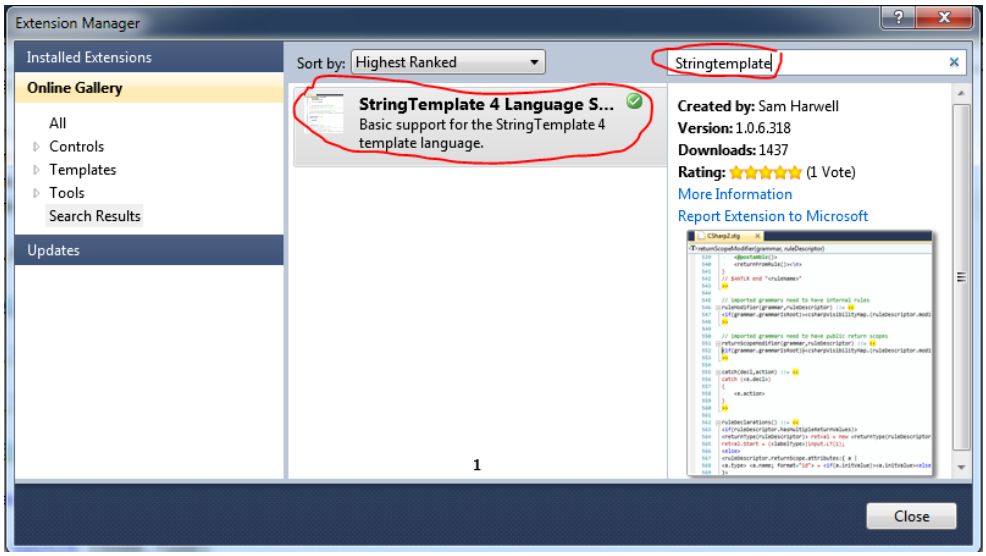
Компоненты интеграции в Visual Studio от Sam Harwell

Самый простой способ установить компоненты, это:

1. Открыть Extension Manager. Для этого в Visual Studio выбрать пункт меню **Tools\Extension Manager…**
2. Выбрать слева раздел **Online Gallery**
3. Набрать в строке поиска **Antlr**



4. Установить расширение ANTLR Language Support
5. Повторить 3 и 4 для **stringtemplate**



6. Перезагрузить Visual Studio.

Примечание. Иногда при первой перезагрузке следуют несколько сообщений о невозможности загрузки некоторых пакетов – из-за конфликтов с другими расширениями. После повторной перезагрузки Visual Studio строкой **devenv.exe /ResetSkipPkgs** сообщения пропадают.

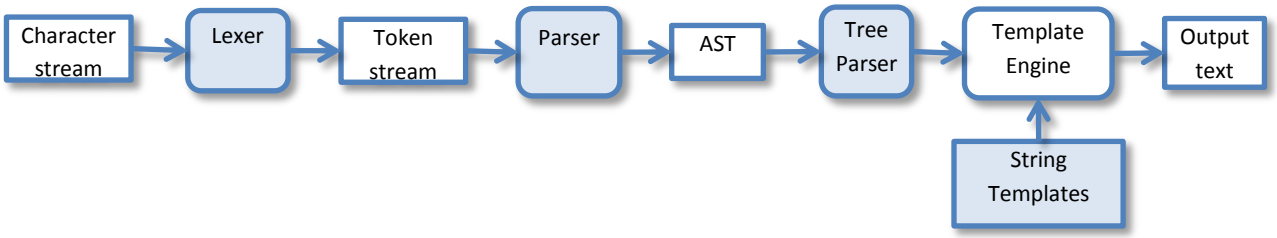
Понятия и подходы работы с ANTLR

В отличие от многих других инструментов создания компиляторов, ANTLR имеет средства для создания не только frontend-части (т.е. лексического и синтаксического анализатора), но и некоторые возможности генерации backend (т.е. генератора кода).

При этом backend может реализовываться весьма по-разному, в зависимости от задач и потребностей.

Вариант 1. Использование механизмов анализатора деревьев и шаблонизатора

На рисунке ниже приведен самый общий вариант, который задействует практически все механизмы ANTLR:



Основные элементы, присутствующие на рисунке (закрашенные фигуры представляют элементы, которые генерирует ANTLR на основе переданных ему описаний грамматик(и)):

- **Character stream** – входной поток символов, содержащий текст программы на исходном языке
- **Lexer** – лексический анализатор, разбивающий исходный поток символов на отдельные лексемы и выполняющий анализ к какому типу принадлежит данная лексема. На выходе лексический анализатор порождает поток токенов (объектов, содержащих информацию о каждой выделенной лексеме и её типе)
- **Token stream** – поток токенов.
- **Parser** – синтаксический анализатор. Просматривает поток токенов и анализирует структуру предложений языка.
- **AST** – абстрактное синтаксическое дерево. AST – это промежуточное представление разобранной программы, удобное для дальнейшего анализа и генерации кода.
- **Tree Parser** – анализатор дерева. Анализатор, используемый для прохода по дереву (в чем-то похож на синтаксический анализатор, но работает не с линейным потоком токенов, а с двухмерной структурой – деревом). Он анализирует структуру и отдельные узлы дерева и вызывает template engine, для генерации выходного текста.
- **Template Engine** – механизм генерации текстов на основе предопределенных шаблонов. Данный механизм формирует выходной текст, составляя его из отдельных фрагментов текста. Каждый такой фрагмент получается на основе заранее определенного шаблона и подставляемых в него параметров
- **String Templates** – шаблоны для генератора вывода (библиотека шаблонов)
- **Output Text** – окончательный текст.

Получается следующая цепочка: поступающая на вход программа разбирается с помощью лексического и синтаксического анализатора, а в результате формируется промежуточное представление программы в виде AST. Затем специальный анализатор деревьев (а их может быть не один – каждый анализатор для своих задач!) проходится по AST и через специальную библиотеку вызывает генерацию выходного текста.

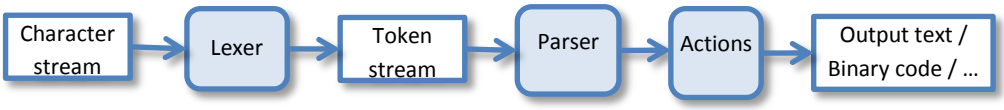
Как видно из описания в данном варианте на входе имеется текст программы, а на выходе – тоже текст (например, на другом языке), т.е. такой вариант в первую очередь предназначен для создания DSL-трансляторов, переводящих программу на DSL, в программу на каком-либо распространенном языке программирования.

Однако, такой подход не единственный, и возможны другие варианты.

Вариант 2. Генерация выхода из действий (семантических правил, actions) в синтаксическом анализаторе

Такой вариант часто встречается в классических учебниках по компиляторам и его поддерживают многие распространённые генераторы компиляторов. В этом варианте действия по генерации результирующего кода встраиваются непосредственно в правила синтаксического анализа.

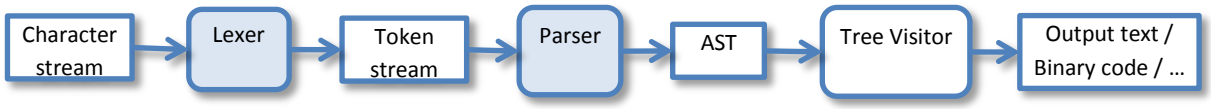
Получается вот такой конвейер обработки:



Действия вызываются синтаксическим анализатором, когда он заканчивает разбирать какую-то грамматическую конструкцию или ее часть. Так как действия это код на произвольном (конечно из тех, что поддерживает ANTLR) языке программирования, то в него может быть встроено, все что угодно: анализ и поиск ошибок, генерация текста или бинарного кода, сбор статистики, …

Вариант 3. Генерация абстрактного синтаксического дерева и его ручной разбор

Этот вариант является промежуточным между 1-ым и 2-ым. С одной стороны синтаксический анализатор здесь также генерирует структуру AST, с другой – для анализа этой структуры используется «рукописный» код.



Обычно для написания кода прохода по полученному AST используют подход называемый Tree Visitor (или hierarchy visitor).

Как и в варианте 2, т.к. код «визитёра» вручную, результатом его выполнения может быть не только текст.

Как будем работать мы?

В дальнейшем, в работе над нашим простым компилятором мы будем ориентироваться на вариант №1. Однако, по мере знакомства с ANTLR будут рассмотрены и оба оставшихся варианта. Таким образом, у нас получится сразу 3 реализации одного и того же компилятора.

Как построить работу с ANTLR?

Примерная схема работы с ANTLR (для варианта 1) может быть представлена так:

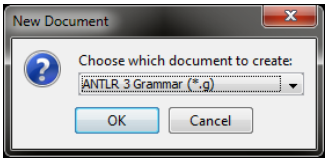
1. Разработка грамматик и отладка их в ANTLRWorks:
    a. разработка грамматик исходного языка (для лексического и синтаксического анализатора)
    b. добавление (по необходимости) действий в правила грамматики
    c. добавление правил генерации AST

2. Разработка грамматики деревьев и создание шаблонов для генерации выхода
3. Генерация кода для всех анализаторов
4. Написание приложения, которое будет вызывать сгенерированные классы

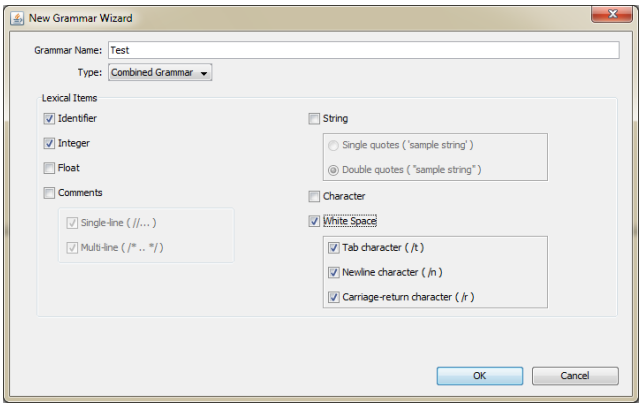
Задание грамматики языка

Для создания простого файла грамматики можно воспользоваться мастером, входящим в состав ANTLRWorks.

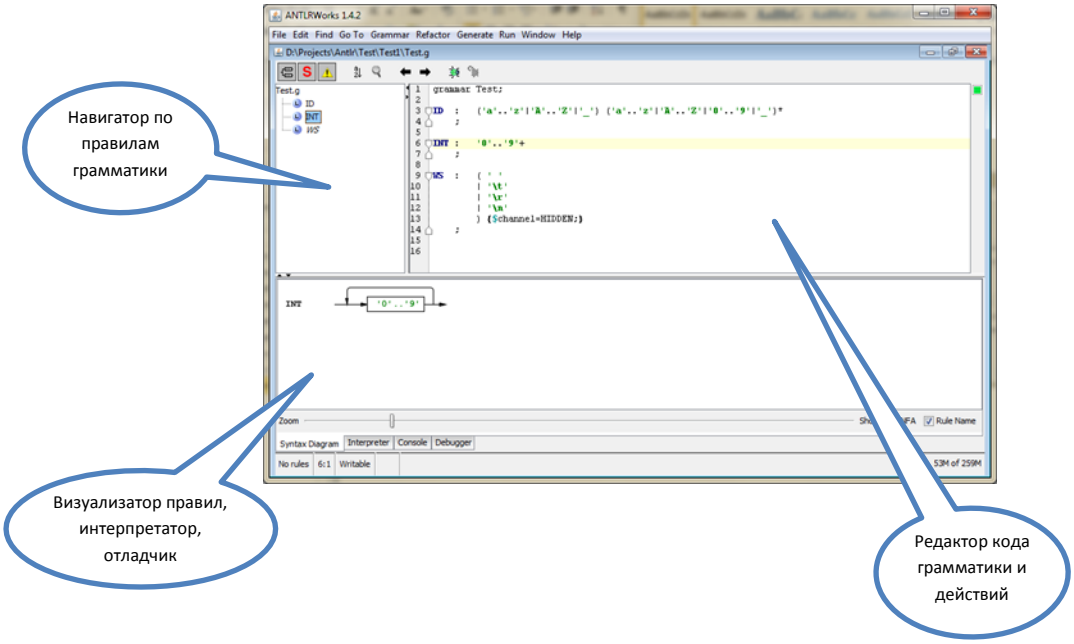
1. Запуск ANTLRWorks (файл antlrworks-1.4.2.jar).
2. Если это первый запуск ANTLRWorks, то мастер создания грамматик запустится автоматически. Если же запуск не первый, необходимо выбрать пункт меню **File\New…**
3. В окне New Document выбрать пункт **ANTLR 3 Grammar (*.g)** (см. рисунок):



4. В окне задания параметров грамматики указать (см. рисунок):
   - имя грамматики (Test)
   - тип грамматики (Combined Grammar)
   - поставить галочки создания лексических правил для:
     - идентификаторов - Identifier
     - целых чисел – Integer
     - пробелов всех типов – White Space



В результате выполнения этих шагов откроется редактор грамматики ANTLRWorks:



Для нас пока наиболее важным будет окно редактора текстов.

К уже существующему коду мы добавим еще одно правило, так, чтобы в результате получилась примерно такая картина:

```
grammar Test;

ID      :       ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
        ;

INT     :       '0'..'9'+
        ;

WS      :       ( ' '
                | '\t'
                | '\r'
                | '\n'
                ) {$channel=HIDDEN;}
        ;

prog
        :       (ID ':=' INT ';') +
        ;
```

Рассмотрим ее более подробно.

Более полное и формальное описание грамматики можно найти в документации на сайте ANTLR, в разделе Grammars.

Основные элементы грамматики

Заголовок грамматики

Первая строчка – заголовок грамматики. В общем виде он задается следующим образом:

```
[тип_грамматики] grammar <имя_грамматики>;
```

Всего ANTLR поддерживает 4 типа грамматик:

- **lexer** – задает правила для лексического анализатора
- **parser** – задает правила для синтаксического анализатора
- **tree** – правила для анализатора AST
- **без явного указания** (как в нашем примере) – задает комбинированную (lexer + parser) грамматику

Обычно в работе используют последние две грамматики. Раздельные грамматики для лексического и синтаксического анализатора бывают полезны когда:

- грамматика слишком большая и с общим файлом работать становится неудобно
- нужно иметь возможность менять лексический блок (например, делать лексические анализаторы с учетом специфики различных ОС) или наоборот использовать один лексический блок с разными с синтаксическими.

Важно! ANTLR требует, чтобы имя грамматики и файла, в котором она описана совпадали.

Строки

Строки (строковые константы) в ANTLR заключаются в одинарные кавычки. Длина не имеет значения, т.е. не различаются «одиночные символы» и «строки».

```
':='
```

Можно использовать escape-последовательности для обозначения специальных символов (большинство из них встречаются в других языках – C++, C#, Java, …):

Символ	Значение
'\n'	Перевод строки
'\r'	Возврат каретки
'\b'	«Забой»
'\f'	Промотка
'\''	Символ одинарной кавычки
'\\'	Символ «наклонная черты»
'\uXXXX'	Символ в Unicode. XXXX – заменяется на шестнадцатеричное представление символа в кодировке Unicode

Комментарии

Бывают одно- и многострочные:

```
// однострочный комментарий

/* многострочный
   комментарий
*/
```

Идентификаторы

Состоят из букв символов, цифр и символа '_', но начинаются всегда с буквы. Для идентификаторов важен регистр первого символа, т.к. по нему происходит разделение правил на правила для лексического и синтаксического анализатора.

Правила

Собственно, правила это основной элемент грамматики (что логично ☺). В самом простом виде синтаксис правил представляет собой конструкцию:

```
[модификатор доступа]
<имя правила>
        :       <альтернатива 1>
        |       <альтернатива 2>
        …
        |       <альтернатива N>
        ;
```

Например

```
LETTER  :       'A'
        |       'B'
        |       'C'
        ;
```

Все правила в комбинированной грамматике делятся на лексические и синтаксические.

Лексические правила используются для описания того, как входящий поток символов разбивается на токены, и служат основной для генерации лексического анализатора. Синтаксические описывают правила построения приложений языка и используются для генерации синтаксического анализатора.

Лексические правила (и соответствующие им имена токенов) должны начинаться с заглавной буквы, а синтаксические – со строчной.

Далее приводятся конструкции, которые можно использовать при задании лексических или синтаксических правил:

Конструкция	Тип анализатора (lexer / parser / any)	Описание
‘string’	any	Сопоставление с указанной строкой в текущей позиции (строка рассматривается как токен, у которого нет предопределенного имени)
’A’..’Z’	lexer	Сопоставление с любым из символов из интервала между ‘A’ и ‘Z’ включительно
(’x’..’y’ | ’a’ | ’b’)	lexer	Сопоставление с любым из символов, перечисленном в выражении
T	any	Для синтаксического анализатора – сопоставление с токеном T Для лексического анализатора – вызов лексического правила T
(A | B | C)	parser	Сопоставление с одним из токенов
r	parser	Вызов на сопоставление правила r

Кроме того, ANTLR поддерживает следующие расширения из нотации EBNF, возможные для любых грамматик.

Оператор (пример использования)	Пояснение
( )	Группировка элементов, к которым применяется оператор
a | b | c	Сопоставление с одной из перечисленных альтернатив
x?	Необязательный элемент
x*	Сопоставление 0 или более раз
x+	Сопоставление 1 или более раз

Перечисленные операторы можно комбинировать.

Например, выражение

```
( ‘0’ | ‘1’ )+
```

описывает строку из одного или более символов 0 и 1 (двоичное число)

В созданном нами в самом начале файле описаны 4 правила:

- **ID**, **INT**, **WS** – лексические правила, описывающие токены «идентификатор», «целое число» и «пробельный символ»
- **prog** – синтаксическое правило, которое описывают программу как «набор одного или более операторов, состоящих из:
    - идентификатора,
    - оператора присваивания ‘:=’
    - целого числа

  и завершающихся точкой с запятой»

Отдельного упоминания заслуживает правило WS, в конце которого можно увидеть строку:

```
{$channel=HIDDEN;}
```

Данная строка представляет собой семантическое действие. О семантических действиях будет сказано позднее, здесь же нужно только пояснить, что токену WS присваивается специальный

признак (можно читать как тип канала - «скрытый»), который говорит синтаксическому анализатору, что нужно игнорировать токены WS, если они встречаются в потоке токенов.

Благодаря этому приёму нам нет необходимости повсеместно вставлять в наши синтаксические правила упоминания пробельных символов. Например, наше правило prog выросло бы до такого:

```
prog : (ID WS* ':=' WS* INT WS* ';' WS*) + ;
```

читаемость, которого, явно оставляет желать лучшего.

Еще один полезный механизм, который может пригодиться при разработке лексических анализаторов, это ключевое слово fragment (добавляется перед именем правила), которое указывает, что данное лексическое правило самостоятельно не распознает никаких токенов и может только вызываться из других лексических правил. Например, правило ID из исходного правила можно описать так:

```
ID : (LETTER | '_') (LETTER | '_' | DIGIT)* ;

fragment LETTER : 'A'..'Z' | 'a'..'z';
fragment DIGIT : '0'..'9';
```

Разработка и отладка грамматики для нашего языка

Разработка грамматики

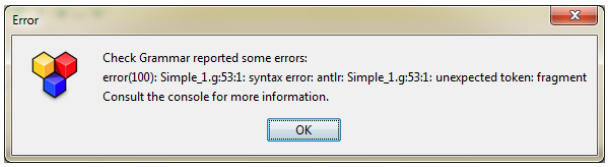
Вооружившись описанными выше понятиями, опишем грамматику для нашего языка. В комментариях к некоторым правилам идет пояснение, упрощающее их понимание.

```
grammar Simple;

// Программа, это непустая последовательность операторов, заканчивающихся точкой с запятой
program : ( stmt ';') + ;

// Оператор, это один из трех операторов: ввода, печати или присваивания
stmt : input_stmt | print_stmt | assign_stmt ;

// Оператор присваивания состоит из идентификатора, знака присваивания и выражения
assign_stmt : IDENT '=' expression ;

// Оператор печати состоит из слова print, за которым следуют одно или более выражений,
// разделённых запятой
print_stmt : 'print' expression (',' expression )* ;

// Оператор ввода состоит из слова input и идентификатора вводимой переменной
input_stmt : 'input' IDENT ;

// Выражение это набор выражений с умножением, разделенных знаками сложения или вычитания
expression : mult ( ('+' | '-') mult)* ;

// Выражение с умножением – это последовательность атомарных единиц, разделенных pyfrfvb
// умножения или деления
mult : atom (( '*' | '/') atom)* ;

atom : IDENT | NUMBER | '(' expression ')' ;

NUMBER : DIGIT + ;

IDENT : (LETTER | '_') (LETTER | '_' | DIGIT)* ;

fragment LETTER : 'A'..'Z' | 'a'..'z' ;

fragment DIGIT : '0'..'9' ;

WS : ('\t' | '\r'? '\n' | ' ')+ { $channel = HIDDEN; } ;
```

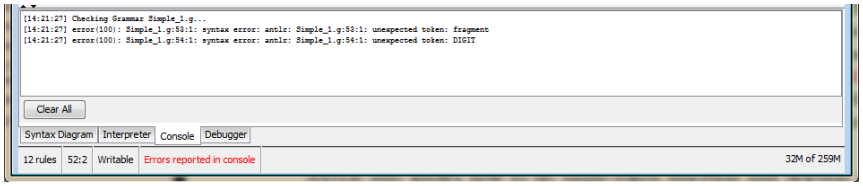
Проверка синтаксиса

После того, как грамматика набрана, можно проверить ее правильность (с точки зрения синтаксиса, но не логики самих правил) непосредственно в ANTLR. Для этого достаточно выбрать пункт меню **Grammar\Check Grammar**. Если проверка прошла успешно, будет выдано сообщение:

В противном случае появится сообщение об ошибке:



Эту же информацию можно увидеть в консоли, внизу редактора (вкладка Console):



Интерпретация и отладка грамматики

Однако, проверка синтаксиса не гарантирует, что сама грамматика распознает именно те конструкции, которые нужно. Поэтому грамматику бывает полезно проверить на различных входных текстах, а в случае обнаружения ошибок, выяснить, в чем именно они состоят, т.е. локализовать и исправить ошибочные правила. Для этого возможны три пути:

- сгенерировать лексический и синтаксический анализатор, написать программу, их использующую и прогнать программу на различных тестах;
- воспользоваться вкладкой Interpreter в ANTLRWorks;
- воспользоваться возможностями отладки, встроенными в ANTLRWorks

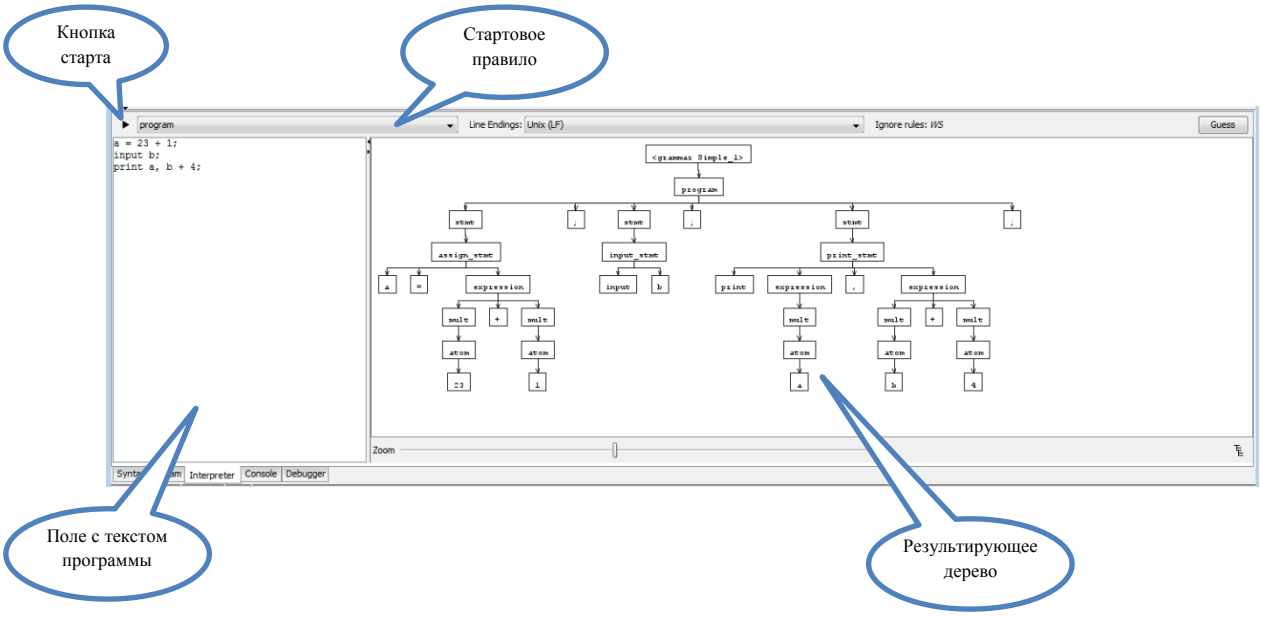
Мы рассмотрим 2 последних, как более простых и наглядных.

Интерпретация грамматики

Интерпретация выполняется встроенными в ANTLRWorks средствами и состоит в построении для указанной грамматики дерева синтаксического разбора. Это дерево показывает: где и какие синтаксические правила применялись, и какие были выделены токены в процессе разбора.

Чтобы запустить интерпретацию, нужно:

- перейти на вкладку Interpreter
- в самом левом поле ввести текст, который требуется разобрать
- в выпадающем списке над ним выбрать правило, с которого будет начат разбор
- нажать кнопку старта интерпретации



Если в процессе разбора произойдет ошибка, то вместо очередного узла будет показан узел с возникшим исключением.

Отладка грамматики

// TODO

Анализатор языка на .Net (C#)

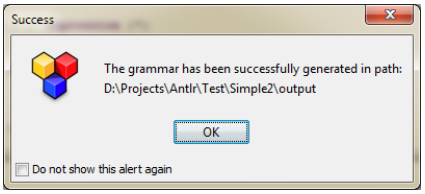
Теперь, когда первая работающая грамматика получена, можно переходить к следующей задаче – создать на ее основе законченную программу, распознающую входящий текст.

Доработка грамматики и генерация

Генерация

Чтобы сгенерировать код классов лексического и синтаксического анализаторов можно воспользоваться одним из следующих способов:

- вызов генерации непосредственно из ANTLRWorks. Для этого выбрать пункт **Generate\Generate Code**. В случае, если генерация пройдет успешно появится соответствующе сообщение:

  При этом надо иметь в виду, что результаты генерации сохраняются в папку, которая указана в настройках ANTLRWorks (меню **File\Preferences**, закладка **General**, поле **Output Path**)
- генерация из командной строки. Если используется порт ANTLR на .Net, то для генерации можно воспользоваться командой:

```
<путь к папке с Antlr>\Antlr3.exe -o <выходная директория> <файл грамматики>
```

Если воспользоваться первым способом, то в выходном каталоге будут созданы файлы с именами:

```
<имя грамматики>Lexer.java
<имя грамматики>Parser.java
<имя грамматики>.tokens
```

если вторым, то:

```
<имя грамматики>Lexer.cs
<имя грамматики>Parser.cs
<имя грамматики>.tokens
```

Такое различие обусловлено, тем что если в грамматике нет явного указания на каком языке генерировать исходный код для анализаторов, то Java-версия ANTLR делает это для языка Java, а .Net-порт – для C#.

Указание целевого языка

Чтобы указать ANTLR для какого языка следует генерировать исходные тексты лексического и синтаксического анализатора, необходимо в текст грамматики добавить раздел опций с опцией **language**.

Опции в грамматике задаются сразу за заголовком перед всеми остальными частями и имеют следующий синтаксис:

```
options { <имя опции 1> = <значение опции 1>;  <имя опции 2> = <значение опции 2>; ... }
```

Полный перечень поддерживаемых опций можно найти в разделе **Grammar options**. Нас пока интересует только опция **language**, которая задает, на каком языке необходимо сгенерировать тексты для лексического и синтаксического анализаторов.

Список известных авторам ANTLR целевых языков приведен на странице Code Generation Targets, правда этот список уже слегка устарел и включает даже не все языки, поддерживаемые ANTLR стандартно.

Для явного указания в качестве целевого языка C# нужно добавить строку:

```
options { language = CSharp3; }
```

Однако, указанная реализация целевого языка (именно реализация CSharp3, для, Java это не применимо) имеет одну важную особенность по сравнению с другими: синтаксические правила, которые будут вызываться кодом извне должны быть явно указаны с атрибутом public.

Более подробно назначение этого атрибута будет рассказано позднее, а здесь приведен код грамматики, полученный после описанных изменений (сами изменения выделены полужирным):

```
grammar Simple;
options { language = CSharp3; }

public
program : ( stmt ';') + ;

stmt : input_stmt | print_stmt | assign_stmt ;

assign_stmt : IDENT '=' expression ;

print_stmt : 'print' expression (',' expression )* ;

input_stmt : 'input' IDENT ;

expression : mult ( ('+' | '-') mult)* ;

mult : atom (( '*' | '/') atom)* ;

atom : IDENT | NUMBER | '(' expression ')' ;

NUMBER : DIGIT + ;

IDENT : (LETTER | '_') (LETTER | '_' | DIGIT)* ;

fragment LETTER : 'A'..'Z' | 'a'..'z' ;

fragment DIGIT : '0'..'9' ;

WS : ('\t' | '\r'? '\n' | ' ')+ { $channel = HIDDEN; } ;
```

После внесения этих изменений необходимо заново сгенерировать код лексера и парсера.

Первая программа-анализатор на базе ANTLR

Наша первая программа с использованием, сгенерированных ANTLR-ом классов будет только проверять корректность введенного текста.

Прежде чем приступать к разработке собственно программы, обсудим, из каких элементов будет состоять последовательность обработки текста в нашей программе. По большому счету мы реализуем усеченный вариант Вариант 2. Генерация выхода из действий (семантических правил, actions) в синтаксическом анализаторе. Основное отличие: у нас не будет генерироваться никаких выходных артефактов, только проверка.

Таким образом, наша программа должна включать:

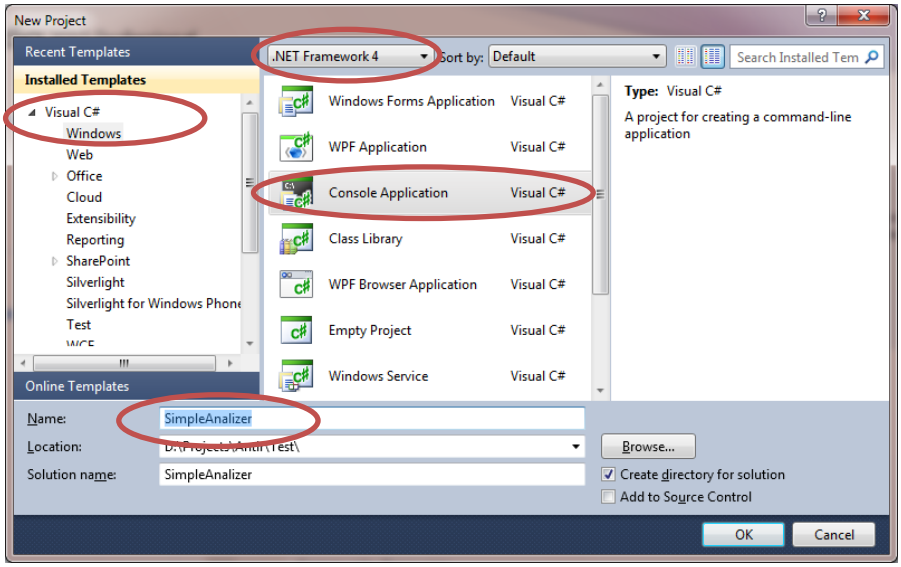
- входящий поток символов (вводимый из файла, с консоли, и т.д.)
- лексический анализатор, разбивающий поток на последовательность токенов
- хранение и передачу потока токенов от лексического анализатора к синтаксическому
- синтаксический анализатор
- печать сообщений о найденных при разборе ошибках

Т.е. схема работы нашей программы можно представить следующим образом:

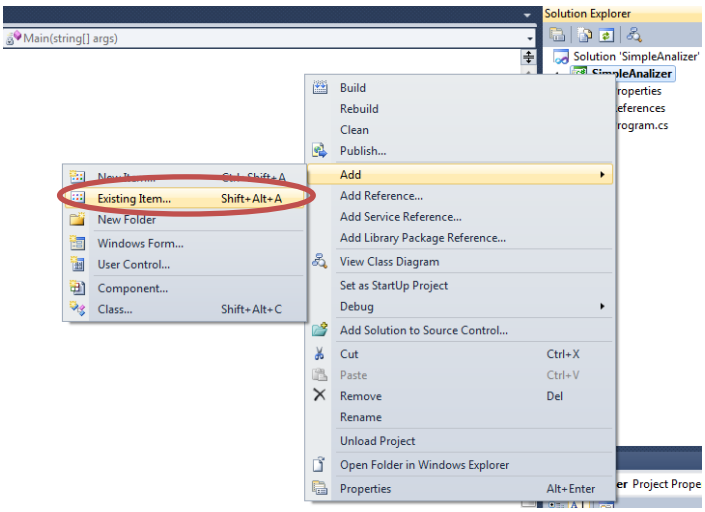
Разработка программы в Visual Studio

Для создания программы по приведенной выше схеме выполним следующие действия:

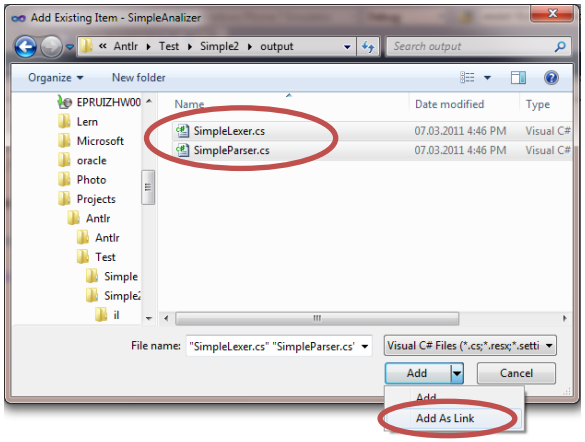
1. Создание консольного проекта C# в Visual Studio. Для этого:
    a. выбрать пункт меню **File\New\Project…**
    b. в окне новых проектов выбрать раздел **Visual C#\Windows**, а в нем пункт **Console Application**
    c. выбрать место создания проекта, его имя (например, SimpleAnalizer),версию .Net Framework (нам подойдет любой, начиная с 3-го).



2. Добавление в проект сгенерированных ANTLR-ом классов.
    a. В окне Solution Explorer щелкнуть правой кнопкой на имя проекта, выбрать **Add\Existing Item…**



b. В открывшемся окне выбрать необходимые файлы (SimpleLexer.cs и SimpleParser.cs), на кнопке **Add** выбрать выпадающий пункт **Add As Link**



В принципе, можно выбрать и вариант Add – в этом случае оба файла будут скопированы в каталог проекта. Это удобно, когда проект нужно куда-то передавать, но не очень удобно, если добавленные файлы будут часто меняться извне (как в нашем случае) – их придется обновлять вручную каждый раз после изменения.

3. Подключить к проекту библиотеку Antlr3.Runtime.dll
4. Заменим содержимое файла Program.cs на следующее:

```
using System;
using Antlr.Runtime;

namespace SimpleAnalizer
{
    class Program
    {
        static void Main(string[] args)
        {
            // Строка с текстом программы - чтобы не связываться с вводом-выводом
            string inputString = "a = 3 + 4; input b; print a + 1, b * 3;";

            /* Создаем цепочку обработки входящей строки
             * - входящий поток, читающий из заранее предопределенной строки
             * - лексический анализатор, подключенный к входному потоку
             * - поток токенов, связанный с ранее созданным лексическим анализатором
             * - синтаксический анализатор, работающий с созданным ранее потоком токенов
             */
            ANTLRStringStream inputSteam = new ANTLRStringStream(inputString);
            SimpleLexer lexer = new SimpleLexer(inputSteam);
```

```
            CommonTokenStream tokenStream = new CommonTokenStream(lexer);
            SimpleParser parser = new SimpleParser(tokenStream);

            // Создаем и подключаем буфер сбора ошибок разбора
            System.IO.StringWriter logBuffer = new System.IO.StringWriter();
            parser.TraceDestination = logBuffer;

            // Вызываем разбор правила program
            parser.program();

            // Анализируем результаты разбора. Если были синтаксические ошибки, выдадим лог анализа
            if (parser.NumberOfSyntaxErrors != 0)
            {
                Console.WriteLine(logBuffer.ToString());
            }
            else
            {
                Console.WriteLine("All ok!");
            }
        }
    }
}
```

Если теперь запустить представленную программу, то в результате на консоли появится сообщение **«All ok!»**, т.к. текст программы, записанный в переменную **inputString** – корректный. Однако, если его заменить, например, на:

```
string inputString = "a = 3 + 4; input b =; print a + 1, b * 3;";
```

то в результате будет выдано сообщение:

```
line 1:20 extraneous input '=' expecting ';'
```

Разработка программы в Visual Studio с использованием компонент от Sam Harwell

Чтобы максимально перенести разработку в Visual Studio, выполним следующие действия (их, к сожалению, придется выполнять для каждого проекта):

1. Создадим новый консольный проект (см. раздел выше)
2. В папке solution-а создадим подпапку Resources\Antlr, куда скопируем содержимое архива **antlr-dotnet-tool-<номер версии>.7z**
3. Файл Antlr3.targets заменим на прилагаемый к пособию (оригинальный файл при компиляции создает сгенерированные файлы в папке bin\Debug или bin\Release проекта, а это не удобно для использования в студии).
4. Выгрузим и откроем на редактирование файл проекта. Для этого:
    a. щелкнем мышью на узле проекта и выберем пункт **Unload Project**.
    b. повторно щелкнем на узле (уже выгруженного проекта) и выберем пункт **Edit <имя файла проекта>**
5. В открытом файле проекта найдем строку:

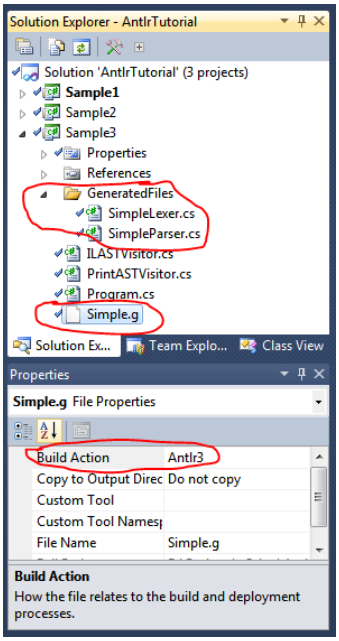
    ```
    <Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
    ```

    и вставим вслед за ней:

    ```
    <PropertyGroup>
      <AntlrBuildTaskPath>$(ProjectDir)..\Resources\Antlr\</AntlrBuildTaskPath>
      <AntlrToolPath>$(ProjectDir)..\Resources\Antlr\Antlr3.exe</AntlrToolPath>
      <AntlrGenCodeFilePath>$(ProjectDir)GeneratedFiles\</AntlrGenCodeFilePath>
    </PropertyGroup>
    <Import Project="$(ProjectDir)..\Resources\Antlr\Antlr3.targets" />
    ```

6. Еще раз щелкнем на узле проекта и выберем пункт **Reload project**.

7. Добавим в проект (не ссылкой, а как существующий элемент) файл грамматики Simple.g
8. В свойствах файла в строке **Build Action** выберем вариант **Antlr3**
9. В раздел References добавим сборку Antlr3.Runtime.dll (она лежит в папке солюшена, в Resources\Antlr)
10. Скомпилируем 1 раз проект.
    a. При этом в папке проекта создастся подпапка GeneratedFiles, содержащая все файлы, сгенерированные Antlr.
11. Добавим в проект все .cs файлы из этой папки, но **как ссылки** (см. предыдущий раздел).
    a. В проекте появится папка GeneratedFiles, содержащая наши сгенерированные лексер и парсер.
12. Поменяем файл Program.cs аналогично предыдущему пункту.



Сгенерированные анализаторы

// TODO Рассмотрим более подробно, что получается в результате работы ANTLR

Атрибуты (attributes) и действия (actions) в грамматиках

В предшествующих разделах мы разработали первый анализатор на основе грамматики, который умеет разобрать текст программы, и ответить на вопрос «Соответствует ли программа грамматике или содержит ошибки?».

Однако только проверка текста – слишком узкая задача. Используя только определение грамматики, мы не сможем решить не только нашу основную задачу – генерацию машинного или IL-кода, но и даже более простую, например, задачу составления карты используемых переменных (какие переменные используются в программе, какого типа, как инициализируются, и т.д.).

Есть несколько способов обработки результатов разбора входной строки. Одним из самых известных является добавление в грамматику семантических атрибутов (attributes) и семантических правил (rules) или действий (actions). Этот способ используется во многих средствах автоматического создания компиляторов, ANTLR – не исключение

Кратко атрибуты и действия можно определить следующим образом:

- **атрибут** – это свойство (переменная), которое, как правило, связано с правилом, и хранит какую-либо его специфическую характеристику. Например, атрибут может хранить текст разобранной строки, связанной с правилом, или глубину вложений оператора if, …
- **действие** – некоторый программный код, срабатывающий в определенные моменты при разборе входной строки (в ANTLR понятие действия трактуется несколько шире: действия – это механизм добавления произвольного кода к генерируемому коду анализаторов, не обязательно связанному с разбором).

Таким образом, атрибуты и действия позволяют разработчику добавлять в процесс разбора произвольную логику на привычных языках программирования. Познакомимся с действиями и атрибутами подробнее.

Действия

Самый распространенный пример использования действий, это некоторые манипуляции в ответ на срабатывание того или иного синтаксического правила (или части правила).

Для демонстрации использования простых действий слегка модифицируем правило разбора операторов:

```
stmt
        :       input_stmt { Console.WriteLine("Input statement!"); }
        |       print_stmt { Console.WriteLine("Print statement!"); }
        |       assign_stmt { Console.WriteLine("Assignment statement!"); }
        ;
```

Теперь по завершении разбора каждого оператора на консоль будет выдаваться сообщение, описывающее, какой именно оператор встретился при разборе. Однако по умолчанию пространство имен System, в котором объявлен класс Console, не добавляется в генерируемые классы (и при компиляции будет выдано сообщение “The name 'Console' does not exist in the current context”). Чтобы решить эту проблему в начало файла грамматики, сразу за объявление опций, мы добавим такую конструкцию:

```
@header { using System; }
```

Т.е. начало файла будет выглядеть как:

```
grammar Simple;
options { language = CSharp3; }
@header { using System; }

public
program : ( stmt ';') + ;
…
```

Если теперь заново сгенерировать анализаторы, перекомпилировать и запустить наш пример, то в результате будет выдано:

```
Assignment statement!
Input statement!
Print statement!
All ok!
```

С точки зрения ANTLR, все, что содержится в фигурных скобках – представляет собой код на целевом языке (том, который указан в опции **language**). ANTLR не проверяет этот код на корректность – это целиком забота программиста.

Обратите внимание, что в фигурные скобки также заключается код семантических предикатов (они используются для проверки каких-либо условий в процессе разбора). Отличить действия и предикаты можно по знаку вопроса, стоящему сразу за закрывающей фигурной скобкой:

```
{isMethodBody}?
```

Некоторые важные замечания по применению действий:

- действия могут находиться в любой части выражения грамматики, при этом порядок выполнения, для, например, такой строки (часть правила):

  ```
  … “строка 1” { // код } “строка 2” …
  ```

  будет следующим:
  - …
  - сопоставление с токеном «строка 1»
  - выполнение кода
  - сопоставление с токеном «строка 2»
  - …
- у каждой альтернативы могут быть свои уникальные действия. Например (часть правила с альтернативой выделена полужирным шрифтом):

  ```
  expression :
          mult (
                  ( '+' { Console.WriteLine("Add"); }
                  | '-' {Console.WriteLine("Sub"); }
                  )
          mult)*
  ```

- операторы +, *, ? распространяются и на действия. При разборе входного текста действие, будет выполнено столько раз, сколько произойдет сопоставлений со строкой. Например, для приведенного в предыдущем пункте правила строка:

  ```
  a - 1 + 1 + 1
  ```

  породит вывод:

```
Sub
Add
Add
```

- если в коде нужно использовать скобки, то нужно следить за тем, чтобы в рамках одного действия все открытые фигурные скобки были корректно закрыты. Два примера (внешние скобки, обрамляющие действие, выделены полужирным):
    - правильный вариант:

      ```
      { { Console.WriteLine("Add"); } }
      ```

    - неправильный вариант (хотя общий баланс скобок и соблюдён, но действие будет разбито на 2):

      ```
      { } Console.WriteLine("Add"); { }
      ```

Действия, в том виде, в каком они приведены в этом разделе, обладают очень существенным недостатком – они практически никак не используют информацию о результатах разбора (максимум – разные действия для разных альтернатив). Для решения этой проблемы используются атрибуты.

Атрибуты

Для демонстрации использования атрибутов вернемся к нашему исходному примеру (до добавления действий) и модифицируем следующее правило:

```
assign_stmt
        : IDENT '=' expression
                { Console.WriteLine("Assign to " + $IDENT.text.ToUpper() +
                        " value of " + $expression.text);
                }
        ;
```

Соберем измененный пример (не забыв добавить `@header { using System; }`) и выполним. В результате получим:

```
Assign to A value of 3 + 4
All ok!
```

В данном примере мы дважды обращались к результатам разбора из действий: в строке **$IDENT.text** (ToUpper() – это уже стандартный метод для строк в .Net Framework) и **$expression.text**.

В общем виде обращение к атрибутам токенов или результатов работы синтаксических правил осуществляется как:

```
$<имя токена или правила>.<имя атрибута>
```

Если нужно обратиться к атрибуту текущего правила (т.е. того, которое сейчас разбирается), то обращение происходит просто в виде:

```
$<имя атрибута>
```

Однако, в этом случае доступны не все атрибуты (для некоторых просто еще не определены значения).

Состав атрибутов для токенов предопределен заранее и не может меняться. Основные атрибуты токенов:

Атрибут	Тип	Комментарий
text	String	Текст лексемы для токена
type	int	Тип токена (все типы кодируются целыми значениями, для которых заводятся именованные константы, которые потом можно использовать в своем коде)
line, pos, index	int	Номер строки и позиция в строке, в которой обнаружен токен А также индекс (номер) токена в потоке токенов.
channel	int	Канал. По большому счету, значение по которому происходит разбиение общего потока токенов на несколько различных. Чаще всего это применяется для того, чтобы исключить часть токенов из обработки. По умолчанию используются каналы **Default** и **Hidden**. Стандартный поток токенов отбрасывает токены, у которых channel == Hidden (см. наш основной пример, токен WS).
tree	Object	Узел AST-дерева, с которым связан данный токен (см. далее). Используется, только если включена генерация AST

В отличие от токенов, атрибуты правил могут расширяться разработчиком. Однако, есть и ряд предопределенных:

Атрибут	Тип	Комментарий
text	String	Текст из входного потока, который был разобран правилом
start, stop	Token	Начальный и конечный токены, из цепочки токенов, которые были разобраны данным правилом
tree	Object	Узел AST-дерева, с которым связан данное правило (см. далее). Используется, если включена генерация AST
st	StringTemplate	Строковый шаблон, сгенерированный для правила (см. далее). Используется, если включен вывод шаблонов

В приведенном примере использование имени токена или правила было однозначным, т.к. каждое имя встречалось ровно 1 раз. Однако, например, вот в таком правиле:

```
if_stmt : 'if' expression 'then' stmt ('else' stmt)? 'endif';
```

правило **stmt** встречается дважды, поэтому, при ссылке на него из действий, возникнет вопрос: о каком из двух операторов идет речь?

Для разрешения этой неоднозначности возможны два подхода:

- размещать действие по обработке каждого из операторов в том месте, где этот оператор встречается – до того, как встретится следующий:

```
if_stmt :
        'if' expression 'then' stmt { // обработка ветки then }
        ('else' stmt { // обработка ветки else } )?
        'endif';
```

- использовать механизм меток для назначения каждому оператору уникального имени:

```
if_stmt :
        'if' expression 'then' then_stmt = stmt
        (else_token = 'else' else_stmt = stmt )?
        'endif'
        {
                Console.WriteLine("then " + $then_stmt.text);
                if ($else_token != null)
                        Console.WriteLine("else " + $else_stmt.text);
                else
                        Console.WriteLine("'else' not present");
```

```
        }
        ;
```

В этом примере каждому оператору **stmt** присваивается уникальная метка (**then_stmt** или **else_stmt**), а также уникальная метка присваивается токену **‘else’**. Последнее сделано для того, чтобы проверить была ли распознана ветка с ‘else’ – если переменная не пустая, то распознавания токена прошло, иначе нет (к сожалению, подобную проверку нельзя сделать с использованием самого правила – в действиях ANTLR только по имени, без указания атрибута, можно обращаться лишь к токенам, но не к правилам).
Вся обработка производится в одном общем действии в конце правила:

- оператор, помещенный в ветку then, печатается всегда
- если встретилось слово ‘else’ то печатается оператор ветки else, иначе выдается сообщение **'else' not present**.

Еще один пример использования меток – это коллекции значений в правилах с операторами * и +. Например, в правиле (слегка модифицированное правило ввода):

```
input_stmt : 'input' IDENT (',' IDENT) *;
```

токен IDENT может встречаться 1 или более раз, причем для нас все идентификаторы равноправны (максимум, что важно – это порядок их следования).

Для такого случая можно воспользоваться списочными метками (т.е. метками, которые хранят не единственное значение, а набирают список всех встретившихся значений). Для нашего случая можно записать такой пример:

```
input_stmt
        :       'input' id+=IDENT (',' id+=IDENT) *
                {
                        Console.WriteLine("Input " + $id.Count);
                        foreach(IToken t in $id)
                        {
                                Console.WriteLine(t.Text);
                        }
                }
        ;
```

Обратите внимание, что в списке набираются объекты типа IToken, у которых свойство Text , в отличие от атрибута text, указывается с прописной буквы.

К сожалению, подобная техника не применима к выражениям, в которых повторяются не токены, а синтаксические правила, например, как в выражении

```
print_stmt : 'print' expression (',' expression )* ;
```

(точнее списочные метки с синтаксическими правилами можно использовать, если грамматика генерирует AST-деревья, но о них будет сказано позже).

Именованные действия

Кроме действий внедряемых непосредственно в правила, в ANTLR существует также механизм, позволяющий добавлять код не к отдельным правилам и альтернативам, а сразу ко всему генерируемому классу или даже в заголовок файла. Ранее мы уже встречались с таким механизмом, когда использовали конструкцию

```
@header { using System; }
```

В общем виде синтаксис именованных действий выглядит следующим образом:

```
@<имя действия> { <код> }
```

ANTLR поддерживает следующие именованные действия уровня грамматики (т.е. распространяющихся на всю грамматику разом) или глобальные действия:

Имя	Описание
header	Добавляет код в самое начало файла (еще до объявления классов анализаторов). Используется для подключения нужных пространств имен (как у нас) или создания вспомогательных классов/структур, которые будут затем использоваться в основном классе
members	Добавляет код в определение классов для анализаторов. Обычно это объявление вспомогательных свойств и методов.
rulecatch	Определяет код, который будет использоваться обработки ошибок распознавания. По умолчанию, если это действие не объявлено, ANTLR реализует стратегию обработки ошибок, при которой: • встреченная ошибка фиксируется в трейсе (см. наш первый пример) • делается попытка исправить ошибку и продолжить разбор Если же данное действие будет объявлено, то код по умолчанию будет заменен на него.
synpredgate	Описывает логическое выражение (т.е. с результатом типа Boolean), которое будет проверяться перед вызовом встраиваемых в правила действий. Данная проверка включается при условии, что установлена опция поддержки отката при разборе (**backtrack = true**) и по умолчанию выражение проверки равно **backtracking==0**

Кроме глобальных именованных действий, существуют также именованные действия на уровне правил. Эти действия называются **init** и **after** и указывают код, который должен выполнится:

- при вызове правила, до начала какого-либо разбора (init)
- после полного завершения разбора и выполнения всех встроенных действий (after)

Синтаксис именованных действий уровня правил аналогичен глобальным правилам, но задаются они не один раз на всю грамматику, а в каждом правиле. Например, модифицировав правило для оператора вывода:

```
print_stmt
        @init {List<string> exp = new List<string>(); }
        @after {
                Console.WriteLine("Print " + exp.Count);
                foreach(String t in exp)
                {
                        Console.WriteLine(t);
                }
        }

        : 'print' exp1=expression { exp.Add($exp1.text); }
                (',' exp2=expression { exp.Add($exp2.text); } )*
        ;
```

мы получим результат, аналогичный тому, что мы чуть ранее получили для оператора ввода, т.е. печать количества операндов у оператора и их значений.

Передача данных между правилами.

Параметры правил и возвращаемые значения

Глобальные и динамические области (scope)

// TODO

Генератор IL-кода на основе действий и атрибутов

Теперь, когда мы знакомы в общих чертах с работой атрибутов действий, можно применить эти знания для решения нашей основной задачи – построения компилятора с языка Simple на IL.

Для того, чтобы упростить разработку кода по генерации IL (чтобы его разрабатывать в Visual Studio, а не ANTLRWorks, который ничего не знает о языке C#), а также упростить разработку и чтение файлов грамматики, которые становится очень сложно читать, если размещать в действиях много кода мы всю кодогенерацию вынесем в отдельный класс Emitter, который будет предоставлять набор методов для генерации тех или иных структур и кода.

Грамматика

```
grammar Simple;
options { language = CSharp3; }

@header { using SimpleCompilator; }

@members {
  Emitter emitter;
  public SimpleParser(ITokenStream input, Emitter emitter)
        : this(input)
  {
    this.emitter = emitter;
  }
}

public
program : (stmt ';') + ;

stmt    : input_stmt | print_stmt | assign_stmt ;

assign_stmt
        :       IDENT '=' expression { emitter.AddAssignStatement($IDENT.text ); }
        ;

print_stmt
        :       'print' expression { emitter.AddPrintStatement(); }
                (',' expression { emitter.AddPrintStatement(); } )*
        ;

expression
        :       mult ( op=('+' | '-') mult { emitter.AddOperation($op.text); } )*
        ;

mult
        :       atom ( op=( '*' | '/') atom { emitter.AddOperation($op.text); } )*
        ;

atom
        :       IDENT { emitter.AddLoadID($IDENT.text); }
        |       NUMBER { emitter.AddLoadConst($NUMBER.text); }
        |       '(' expression ')'
        ;

input_stmt
        :       'input' IDENT { emitter.AddInputStatement($IDENT.text ); }
        ;


NUMBER  :       DIGIT + ;

IDENT   :       (LETTER | '_') (LETTER | '_' | DIGIT)* ;

fragment LETTER :       'A'..'Z' | 'a'..'z';

fragment DIGIT  :       '0'..'9';

WS      :       ('\t' | '\r'? '\n' | ' ')+ { $channel = Hidden; };
```

Класс-эмиттер

```
using System.Collections.Generic;
using System.Text;
using System.IO;

namespace SimpleCompilator
{
    public class Emitter
    {
        /// Класс для хранения информации о переменных
        public class VariableInfo
        {
            public VariableInfo(string name)     {  Name = name; }
            public string Name { get; set; }
        }

        /// Таблица переменных
        IDictionary<string, VariableInfo> variableTable = new Dictionary<string, VariableInfo>();

        /// Буфер для формирования тела основного метода
        StringBuilder methodBody = new StringBuilder();

        /// Запись в выходной поток шапки файла
        void WriteHeader(StreamWriter outWriter)
        {
            // Объявление сборки, модуля и подключаемых сборок
            outWriter.WriteLine(".assembly Program { }");
            outWriter.WriteLine(".module Program.exe");
            outWriter.WriteLine(".assembly extern mscorlib { }");
            outWriter.WriteLine();

            // Объявление основного метода - точки входа (стек ставим условно)
            outWriter.WriteLine(".method public static void Main() {");
            outWriter.WriteLine(".entrypoint");
            outWriter.WriteLine(".maxstack 300");
        }

        /// Запись окончания файла
        void WriteFooter(StreamWriter outWriter)
        {
            outWriter.WriteLine("ret");
            outWriter.WriteLine("}");
        }

        /// Запись объявления всех встреченных локальных переменных
        void WriteLocals(StreamWriter outWriter)
        {
            if (variableTable.Count == 0) return;

            StringBuilder localsString = new StringBuilder();
            localsString.Append(".locals (");
            foreach (VariableInfo variable in variableTable.Values)
            {
                localsString.AppendFormat("int32 {0},", variable.Name);
            }
            localsString.Remove(localsString.Length - 1, 1);
            localsString.Append(")");

            outWriter.WriteLine(localsString.ToString());
        }

        /// Запись ранее сгенерированного тела метода
        void WriteMethodBody(StreamWriter outWriter)
        {
            outWriter.WriteLine(methodBody.ToString());
        }

        /// Запись всего выходного файла
        public void SaveMSIL(string fileName)
        {
            StreamWriter outWriter =
                new StreamWriter(File.Create(fileName), new System.Text.UTF8Encoding(true));

            WriteHeader(outWriter);
            WriteLocals(outWriter);
            WriteMethodBody(outWriter);
            WriteFooter(outWriter);
```

```
            outWriter.Flush();
        }

        /// Добавление кода для оператора присваивания
        public void AddInputStatement(string variableName)
        {
            if (!variableTable.Keys.Contains(variableName))
            {
                variableTable.Add(variableName, new VariableInfo(variableName));
            }

            methodBody.AppendLine("ldstr \"Введите значение переменной " + variableName + ": \"");
            methodBody.AppendLine("call void [mscorlib]System.Console::Write(string)");

            methodBody.AppendLine("call string [mscorlib]System.Console::ReadLine()");
            methodBody.AppendLine("call int32 [mscorlib]System.Int32::Parse(string)");
            methodBody.AppendLine("stloc  " + variableName);

        }

        /// Добавление кода для оператора печати (только целые значения)
        /// Здесь формируется только операция вывода - само значение в этот момент уже в стеке
        public void AddPrintStatement()
        {
            methodBody.AppendLine("call void [mscorlib]System.Console::WriteLine(int32)");
        }

        /// Добавление кода для оператора присваивания
        /// Аналогично оператору печати здесь формируется только код загрузки переменной
        /// - значение уже в стеке
        public void AddAssignStatement(string variableName)
        {
            if (!variableTable.Keys.Contains(variableName))
            {
                variableTable.Add(variableName, new VariableInfo(variableName));
            }

            methodBody.AppendLine("stloc  " + variableName);
        }

        /// Загрузка в стек значения локальной переменной
        public void AddLoadID(string variableName)
        {
            methodBody.AppendLine("ldloc  " + variableName);
        }

        /// Загрузка в стек константы
        public void AddLoadConst(string number)
        {
            methodBody.AppendLine("ldc.i4  " + number);
        }

        /// Генерация кода операций
        public void AddOperation(string op)
        {
            switch (op)
            {
                case "+":
                    methodBody.AppendLine("add");
                    break;
                case "-":
                    methodBody.AppendLine("sub");
                    break;
                case "*":
                    methodBody.AppendLine("mul");
                    break;
                case "/":
                    methodBody.AppendLine("div");
                    break;
                default:
                    break;
            }
        }
    }
}
```

Код основной программы

```
using System;

namespace SimpleCompilator
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length == 2)
            {
                Antlr.Runtime.ANTLRFileStream inStream = new Antlr.Runtime.ANTLRFileStream(args[0]);
                SimpleLexer lexer = new SimpleLexer(inStream);

                Emitter emitter = new Emitter();

                Antlr.Runtime.CommonTokenStream tokenStream = new Antlr.Runtime.CommonTokenStream(lexer);
                SimpleParser parser = new SimpleParser(tokenStream, emitter);

                parser.program();
                emitter.SaveMSIL(args[1]);
            }
            else
            {
                Console.WriteLine("usage: <program> <inputfile> <outputfile>");
            }
        }
    }
}
```

Введение в абстрактные синтаксические деревья (AST)

Предыдущий пример, использующий для генерации действия и атрибуты, обладает рядом весьма существенных недостатков:

- **смешение декларативного (описание грамматики) и императивного (сам действия) кода**. Такой код довольно сложно поддерживать и отлаживать. Мало того, смешиваются сразу несколько фаз: разбор и компиляция – что еще более усугубляет проблему поддержки кода.
- **работа только с текущей разобранной частью**. Как следствие, нет простой возможности выполнить обработку результатов анализа в несколько проходов (например, сначала собрать информацию обо всех переменных и их области видимости, а затем сгенерировать исполнимый код) – приходится усложнять код генерации или так видоизменять язык, чтобы все информацию можно было получить и обработать за один проход

Для решения означенной проблемы давно и успешно применяют двухстадийную компиляцию:

- разбор исходного языка и сохранение результата в виде некоего, удобного для дальнейшей работы промежуточного представления
- выполнение стадии построения (которая может включать не только получение исполнимого кода, но и всевозможные оптимизации) на базе промежуточного представления

Способов промежуточного представления придумано множество. Есть линейные (например, трехадресный код), а есть графовые (синтаксические деревья, графы потоков контроля). Последние, конкретно абстрактные синтаксические деревья в последнее время стали одними из наиболее популярных методик промежуточного представления.

Синтаксические и абстрактные синтаксические деревья

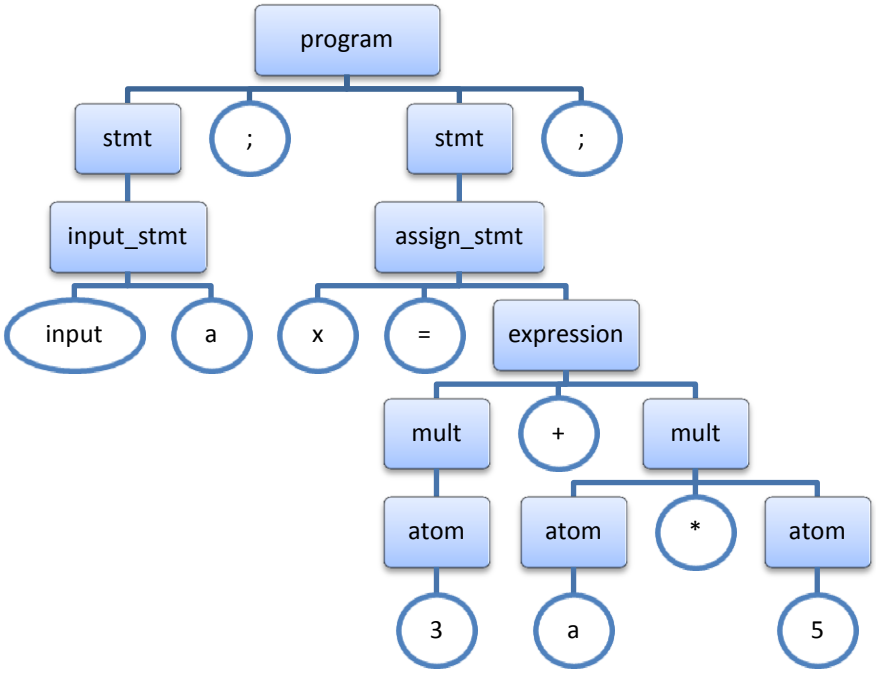
Синтаксическим деревом, а точнее **деревом разбора** называется дерево, структура которого повторяет процесс сопоставления исходного текста и грамматики. Такие деревья обычно записывают следующим образом:

- внутренние узлы представляют правила грамматики;
- листья представляют токены из исходного текста;

В частности, для нашей исходной грамматики и входного и вот такого исходно текста:

```
input a;

x = 3 + a * 5;
```

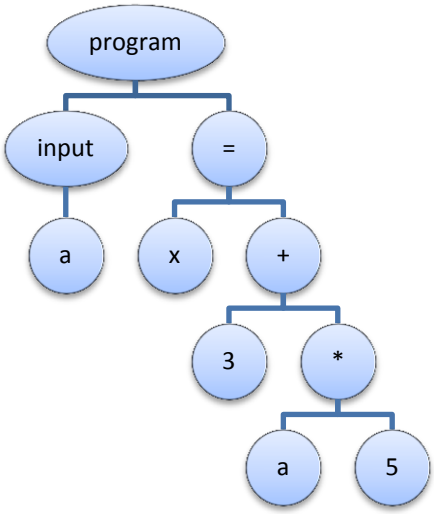
Дерево синтаксического разбора будет иметь следующий вид (прямоугольниками выделены правила, окружностями/овалами – токены):



Даже беглый анализ показывает, что в нем содержится множество избыточных элементов, например:

- токены «;», которые в линейном тексте разделяли операторы, и были крайне важны при анализе, теперь абсолютно не при деле – операторы и так каждый в своем поддереве;
- правило «stmt», которое мы вводили для удобства записи грамматики, не несет никакой смысловой нагрузки – в каждом поддереве за ним идет правило конкретного оператора;
- у правил «input_stmt» и «assign_stmt» есть токены «input» и «=», которые не добавляют новой информации (мы и так уже знаем, что это за операторы);
- правила «expression», «mult» и «atom» нужны нам только для задания приоритетов операторов (чтобы расположить операторы правильно в дереве), и дополнительной информации также не несут.

Если принять во внимание все указанные замечания, то можно перестроить дерево следующим образом:



Дерево стало на много проще для восприятия, но не потеряла никакой значимой информации.

Такое дерево называют **абстрактным синтаксическим деревом** (abstract syntax tree, AST), т.е. дерево, представляющее синтаксис абстрактного (в противовес конкретному) языка – языка содержащего только значимые элементы.

Во многих классических компиляторах компиляторов (например bison) специальных средств для работы с AST не предусмотрено – разработчик, если это необходимо строит дерево сам в коде действий (actions). Однако, в ANTLR имеются сразу несколько механизмов для работы с AST, от генерации до обработки.

Начнем как водится с генерации…

Генерация AST в ANTLR

В ANTLR можно условно выделить три метода генерации AST на основе грамматики разбора («условно», потому что эти методы можно смешивать):

- конструирование AST по умолчанию
- с использованием операторов конструирования деревьев
- с использованием правил замены (Rewrite Rules)

Конструирование AST по умолчанию

Для того чтобы включить данный режим не требуется практически никаких действий. Достаточно лишь добавить опцию

```
options
{
	output = AST;
}
```

Алгоритм генерации такого дерева довольно прост:

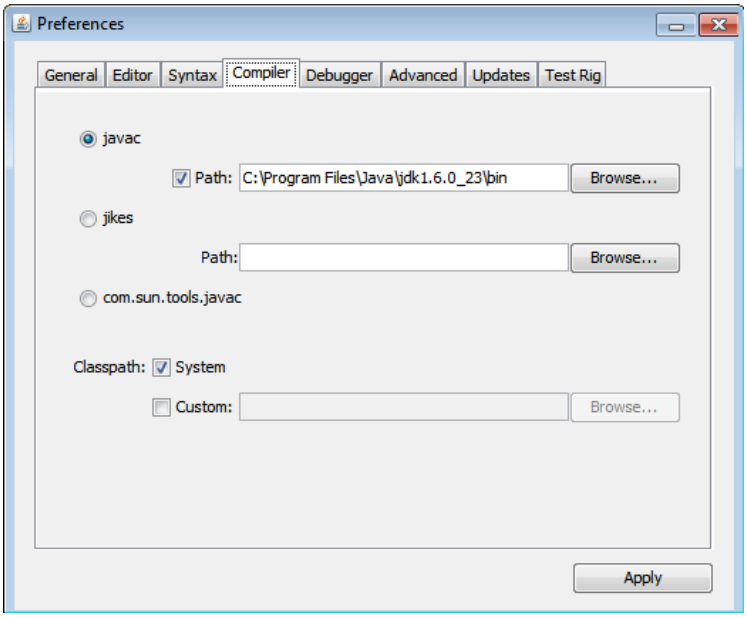
1. При старте разбора любого правила создается корень нового поддерева. Пустой – т.е. сами правила в дерево не добавляются!!!
2. Все разобранные токены добавляются как дочерние элементы к этому корню, а для всех встреченных правил вызывается генерация нового поддерева (как в п. 1)
3. По завершении разбора правила, сформированное им дерево добавляется в качестве поддерева к вышележащему правилу. Если у дерева был пустой корень, то все его дочерние элементы добавляются непосредственно к родительскому правилу.

По сути, у нас получается, что все токены будут добавлены к одному корневому узлу.

Чтобы проверить полученный результат, сделаем следующее:

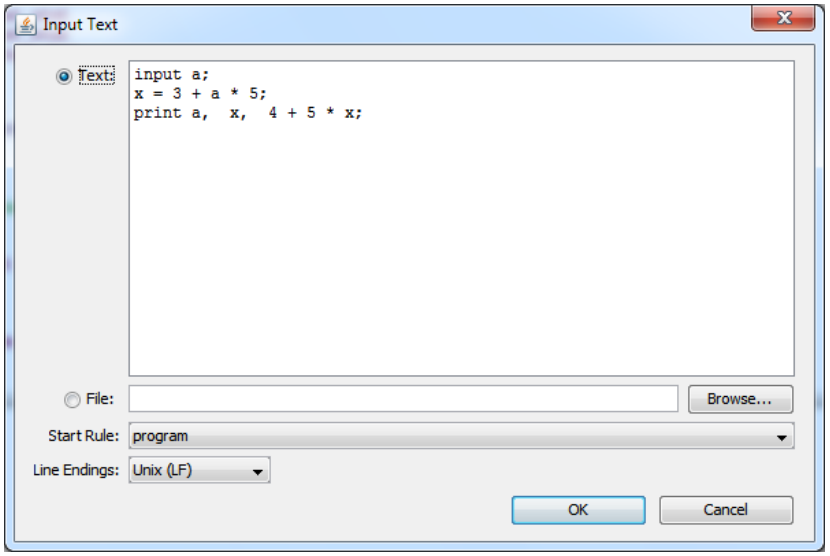
- настроим отладку грамматики в ANTLRWorks.
  Для этого:
  - скачаем и установим JDK (например, отсюда http://www.oracle.com/technetwork/java/javase/downloads/jdk-6u31-download-1501634.html)
  - запустим ANTLWorks, откроем его настройки (**File\Preferences**) и перейдем на закладку **Compiler**

- выберем опцию javac и укажем путь к компилятору (это подпапка bin, той папки, куда был установлен JDK)

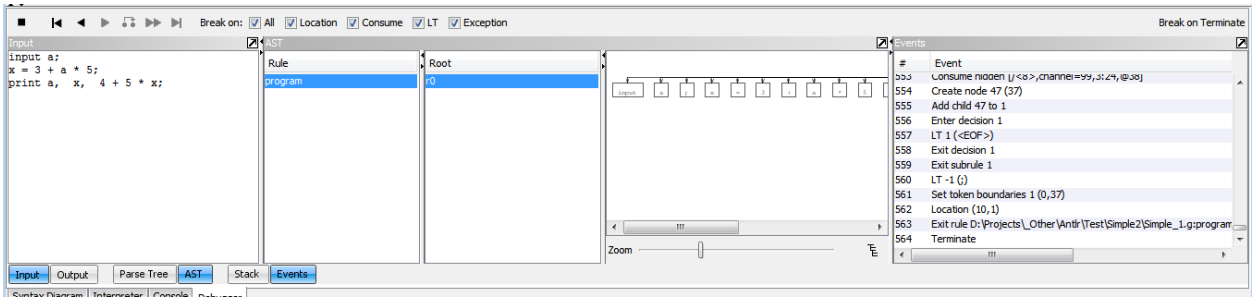


- доработаем грамматику:
  - откроем исходную нашу грамматику (до «заточки» ее под генерацию кода на C#) в ANTLRWorks
  - добавим опцию output = AST (см. выше)
- запустим отладку
  - выберем пункт **Run\Debug**
  - введем текст примера для отладки:

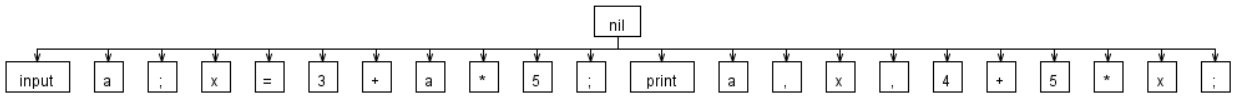
```
input a;
x = 3 + a * 5;
print a, x, 4 + 5 * x;
```



- в нижней части окна переключимся на закладку Debugger, выберем режим показа AST и управляя клавишами отладки просмотрим результат:



Наше дерево выглядит так (один общий корень, и привязанные к нему терминалы):



Понятно, что работать с таким практически линейным представлением нам неудобно – оно и близко не похоже на пример AST, который мы рассматривали.

Конструирование AST с использованием операторов конструирования деревьев

Второй по сложности способ конструирования деревьев состоит в использовании двух специальных операторов «!» и «^». Эти операторы ставятся после терминалов или правил в исходной грамматике и влияют на то, как будет строиться дерево для текущего разбираемого правила:

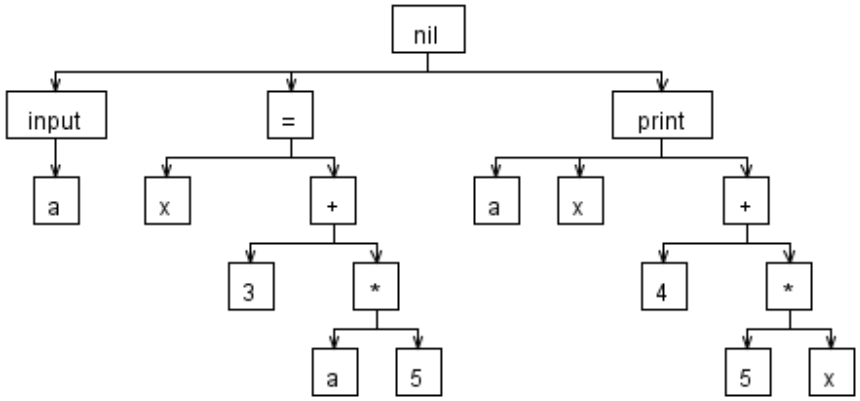
Оператор	Описание
!	Не добавлять текущий элемент в общее AST
^	Сделать помеченный элемент корнем текущего разбираемого правила. Все остальные элементы в текущем правиле становятся его детьми. Если же в одном правиле несколько операторов ^, то построение идет следующим образом: • все элементы до и после первого элемента с оператором «^»являются его детьми • как только встречается второй элемент с «^», то: ○ он становится новым корнем ○ построенное к этому моменту дерево (в этом дереве корень первый элемент с «^») становится первым дочерним элементом в новом дереве ○ все последующие элементы без «^» становятся детьми нового корня

Вот пример для нашей исходной грамматики:

```
grammar Simple;

options { output = AST; }

public program  : ( stmt ';'!) +;

stmt : input_stmt | print_stmt | assign_stmt ;

assign_stmt : IDENT '='^ expression ;

print_stmt : 'print'^ expression (','! expression )* ;

input_stmt : 'input'^ IDENT ;

expression : mult ( ('+' | '-')^ mult)* ;
```

```
mult : atom (( '*' | '/')^ atom)* ;

atom : IDENT | NUMBER | '('! expression ')'! ;

NUMBER : DIGIT + ;

IDENT : (LETTER | '_') (LETTER | '_' | DIGIT)* ;

fragment LETTER : 'A'..'Z' | 'a'..'z' ;

fragment DIGIT : '0'..'9' ;

WS : ('\t' | '\r'? '\n' | ' ')+ { $channel = HIDDEN; } ;
```

Посмотрим, какое дерево будет выдано для нашего примера:



Получилось очень похоже на то, к чему мы стремились!

Конструирование AST с использованием правил замены (Rewrite Rules)

Операторы конструирования дают уже вполне приличный результат, однако, у них есть ряд ограничений:

- они могут манипулировать только токенами, которые есть в самом правиле (т.е. не могут добавить новые)
- они не могут менять порядок токенов

Для нашей задачи, как видно из предыдущего примера, это не существенно, но для более сложных языков может быть серьезным ограничением. Для обхода этих ограничений ANTLR предлагает еще один механизм – **правила замены (rewrite rules)**.

В самом общем виде правила замены имеют следующий синтаксис:

```
<имя_правила_грамматики>:
                        <альтернатива_1> -> <выражение_замены_для_альтернативы_1>
                      | <альтернатива_2> -> <выражение_замены_для_альтернативы_2>
                            ...
                      | <альтернатива_N> -> <выражение_замены_для_альтернативы_N>
                            ;
```

Выражения замен описывают вид синтаксического дерево, которое оно примет, если сработает альтернатива. Сама нотация для описания деревьев имеет вид:

```
^(root child1 child2 ... childN)
```

Первый элемент внутри скобок – корень, каждый следующий – его потомок. При этом любой из элементов (кроме корня), может быть как конечным элементом, так и новым поддеревом. В последнем случае синтаксис у поддерева такой же, как у самого дерева.

В качестве демонстрации приведем пример дерева, которое получилось в предыдущем пункте:

```
^(nil ^(input a) ^(= x ^(+ 3 ^(* a 5)) ^(print a x ^(+ 4 ^(* 5 x))))
```

Вот несколько примеров правил замены:

1. Удаление ненужных токенов:

```
atom : '(' expression ')' -> expression;
```

2. Переупорядочивание токенов (и удаление лишних)

```
decl : 'var' ID ':' type -> type ID ;
```

3. Создание поддерева с заранее неизвестным количеством узлов:

```
list : ID (',' ID)* -> ID+ ;

decl : 'int' ID (',' ID)* -> ^('int' ID+) ;
```

Одной из интересных и удобных особенностей правил замены является возможность использования так называемых **мнимых узлов** (imaginary nodes). Мнимые узлы – это просто именованные константы, которые можно использовать взамен токенов, которые встречаются в исходном языке.

Например, вместо того, чтобы использовать в AST строку ‘print’ (ключевое слова оператора печати), можно ввести константу STMT_PRINT. В результате мы уменьшим вероятность ошибки (если слово ‘print’ встречается многократно, то однажды при его наборе мы можем просто ошибиться).

Наша конечная грамматика может теперь выглядеть так:

```
grammar Simple;
options { output = AST; }

tokens {        PROGRAM; STMT_ASSIGN; STMT_PRINT; STMT_INPUT;
                PLUS = '+'; MINUS = '-'; MUL = '*'; DIV = '/'; }

public program  : (stmt ';') + -> ^(PROGRAM stmt+);

stmt    : input_stmt | print_stmt | assign_stmt ;

assign_stmt : IDENT '=' expression -> ^(STMT_ASSIGN IDENT expression);

input_stmt : 'input' IDENT -> ^(STMT_INPUT IDENT);

print_stmt : 'print' expression (',' expression )* -> ^(STMT_PRINT expression *) ;

expression : mult ( (PLUS | MINUS )^ mult)* ;

mult : atom (( MUL | DIV)^ atom)* ;

atom :  IDENT | NUMBER
        | '(' expression ')' -> expression ;

NUMBER : DIGIT +;

IDENT : (LETTER | '_') (LETTER | '_' | DIGIT)* ;

fragment LETTER : 'A'..'Z' | 'a'..'z' ;

fragment DIGIT : '0'..'9' ;

WS : ('\t' | '\r'? '\n' | ' ')+ { $channel = Hidden; };
```

Обработка AST

Как уже говорилось ранее, одна из задач создания AST – упрощение обработки результатов. В чем же заключается это упрощение?

Основная идея состоит в том, что после построения дерева мы знаем все типы входящих в дерево узлов: каково содержание корневых и дочерних элементов (ведь мы явно задавали правила формирования дерева!).

Например (следуя нашей предыдущей грамматике) мы знаем, что:

- узел «программа» содержит некоторое количество (от одного) детей, каждый из которых:
  - либо оператор присваивания
  - либо оператор ввода
  - либо оператор вывода
- узел «оператор присваивания» содержит ровна 2 потомка:
  - имя переменной куда присвоить
  - выражение которое нужно присвоить
- узел «оператор ввода» содержит 1 потомка:
  - переменную куда присвоить введенное значение
- …

Думаю, идея обработки в целом вырисовывается:

- мы пишем процедуры, каждая из которых обрабатывает (обходит) свой тип узла
- процедура может вызывать другие процедуры для обработки своих потомков (в зависимости от типа каждого), агрегировать и обрабатывать результат обхода всего поддерева

Осталось понять, какое программное API для доступа к готовым AST предоставляет ANTLR. Здесь все довольно прозрачно:

- все узлы дерева это экземпляры класса **CommonTree**, который реализует интерфейс **ITree**
- интерфейс ITree в частности включает такие свойства как:
  - **ChildCount** – число детей
  - **Children** – полный список детей
  - **Type** – число, которое описывает токен, который является корнем в этом дереве. Тут следует оговориться, что каждому использованному в грамматике токену ANTLR присваивает некий внутренний числовой идентификатор.
    К сожалению, если мы не используем мнимые узлы, то данное значение нам практически бесполезно (т.к. никто не гарантирует, какие именно числа будут присвоены обычным токенам). Однако, в случае мнимых узлов – мы вполне можем ими воспользоваться.
- чтобы получить доступ к корневому узлу всего разобранного дерева:
  - запустить разбор обычно, т.е. вызвав стартовый метод парсера
  - сохранить результат, который вернет этот метод (скорее всего это будет класс **AstParserRuleReturnScope** или его потомок)
  - обратиться к свойству Tree данного класса

В качестве примера рассмотрим «обходчик», который просто по-русски интерпретирует встреченные им узлы:

```
namespace Sample3
{
    using System;
    using System.Linq;

    using Antlr.Runtime.Tree;

    class PrintASTVisitor
    {
        public void visitProgram(CommonTree tree)
        {
            foreach (CommonTree node in tree.Children.OfType<CommonTree>())
            {
                switch (node.Type)
                {
                    case SimpleParser.STMT_ASSIGN:
                        this.visitAssignStatement(node);
                        break;

                    case SimpleParser.STMT_INPUT:
                        this.visitInputStatement(node);
                        break;

                    case SimpleParser.STMT_PRINT:
                        this.visitPrintStatement(node);
                        break;

                    default:
                        Console.WriteLine("Error in PROGRAM node!");
                        break;
                }
            }
        }

        void visitAssignStatement(CommonTree tree)
        {
            var variable = tree.GetChild(0) as CommonTree;
            var expression = tree.GetChild(1) as CommonTree;

            Console.WriteLine("Присваиваем переменной " + variable.Token.Text +
                " выражение " + this.visitExpression(expression));
        }

        void visitPrintStatement(CommonTree tree)
        {
            string allExpressions = string.Empty;

            foreach (CommonTree node in tree.Children.OfType<CommonTree>())
            {
                allExpressions += this.visitExpression(node) + " ";
            }

            Console.WriteLine("Печатаем значения " + allExpressions);
        }

        void visitInputStatement(CommonTree tree)
        {
            Console.WriteLine("Вводим значение переменной " +
                     (tree.GetChild(0) as CommonTree).Token.Text);
        }

        string visitExpression(CommonTree tree)
        {
            switch (tree.Type)
            {
                case SimpleParser.PLUS:
                case SimpleParser.MINUS:
                case SimpleParser.MUL:
                case SimpleParser.DIV:
                    var leftExp = tree.GetChild(0) as CommonTree;
                    var rightExp = tree.GetChild(1) as CommonTree;

                    return "(" + this.visitExpression(leftExp) + tree.Token.Text +
```

```
                    this.visitExpression(rightExp) + ")";

                case SimpleParser.NUMBER:
                case SimpleParser.IDENT:
                    return tree.Token.Text;

                default:
                    return string.Empty;
            }
        }
    }
}
```

А вот как выглядит его вызов:

```
using Antlr.Runtime;
using Antlr.Runtime.Tree;

namespace Sample3
{
    class Program
    {
        static void Main(string[] args)
        {
            var inStream = new Antlr.Runtime.ANTLRFileStream(args[0]);
            var lexer = new SimpleLexer(inStream);

            var tokenStream = new CommonTokenStream(lexer);
            var parser = new SimpleParser(tokenStream);

            var tree = parser.program().Tree as CommonTree;

            var visitor = new PrintASTVisitor();
            visitor.visitProgram(tree);

        }
    }

}
```

Грамматика для деревьев. Текстовые шаблоны

Генерация на основе шаблонов