МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ»

Основы синтаксического разбора, построение синтаксических анализаторов

Учебно-методическое пособие для вузов

Составители:
Д.И. Соломатин,
А.В. Копытин
А.И. Другалев

Воронеж

2014

Утверждено ученым советом факультета компьютерных наук от 17 сентября 2014 г., протокол № 13

Рецензент кандидат технических наук, доцент кафедры технологий обработки и защиты информации Гаршина В.В.

Учебно-методическое пособие подготовлено на кафедре программирования и информационных технологий факультета компьютерных наук Воронежского государственного университета.

Рекомендуется для студентов дневного отделения факультета компьютерных наук при изучении дисциплины «Теория компиляторов».

Для направлений: 230400 – Информационные системы и технологии
231000 – Программная инженерия
010200 – Математика и компьютерные науки

1. Основы синтаксического анализа

1.1. Этапы анализа

В работе любого *транслятора* или *интерпретатора* присутствует фаза разбора входной программы, представленной, как привило, в виде текста. Задача транслятора – преобразовать одно представление программы в эквивалентное другое представление. *Компиляторы* – частный случай трансляторов, которые переводят программу в представление, понятное компьютеру – *ассемблерный код* или *байт-код*. Задача интерпретатора – выполнить входную программу без преобразования в другое представление.

Грань между трансляторами и интерпретаторами достаточно условна и размыта, так как современные интерпретаторы перед интерпретацией обязательно переводят входную программу в какое-либо промежуточное представление, удобное для интерпретации (в частных случаях в качестве такого промежуточного представления может быть непосредственно код целевой платформы, т.е. по сути выполняется компиляция кода «на лету»).

Также следует уточнить, что под входной программой, о которой говорится в предыдущих абзацах, не следует узко понимать исключительно код на языке программирования, в частных случаях это могут быть, например, данные, представленные в текстовых форматах XML, JSON и т.д.

Процесс разбора входной программы, часто неформально называемый *парсингом*, как правило, состоит из двух этапов: *лексического анализа* и *синтаксического анализа* (стоит отметить, что существуют подходы к парсингу, в которых этап лексического анализа явно не выделен).

На этапе лексического анализа входная строка из последовательности символов переводится в последовательность *лексем* (*токенов*), суть этого преобразования показана на Рис. 1.

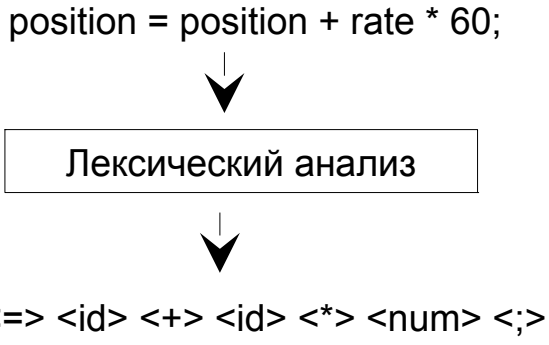

Рис. 1. Лексический анализ

Из Рис. 1. видно, что суть лексического анализа заключается, если говорить просто, в выделении из последовательности «букв» входной

программы последовательности «слов», неделимых с точки зрения последующего анализа. Эти «слова», называемые лексемами или токенами, делятся на классы. В примере на Рис. 1. после преобразования получены лексемы шести классов (<id> – идентификатор, <num> – число, <=> – оператор присвоения, <*> – оператор сложения, <*> – оператор умножения и <;> – «разделитель»). К лексемам могут быть привязаны различные атрибуты, в частности, фрагмент текста входной строки, из которого была получена данная лексема, позиция этого фрагмента в тексте и т.д.

На этапе синтаксического анализа линейная последовательности лексем сопоставляется с неким формальным набором правил построения предложений *естественного* или *формального* языка (далее в данном пособии речь пойдет исключительно о формальных языках, наиболее яркими примерами которых являются языки программирования).

В простейшем случае синтаксический анализ позволяет получить ответ, принадлежит ли входная строка языку, с которым мы ее пытаемся сопоставить или нет. Однако на практике одного такого ответа (принадлежит или нет) недостаточно и, как правило, параллельно с синтаксическим разбором происходят какие-либо действия, направленные на дальнейшее преобразование входной строки (на этапе лексического анализа уже не просто входной строки, а последовательности лексем).

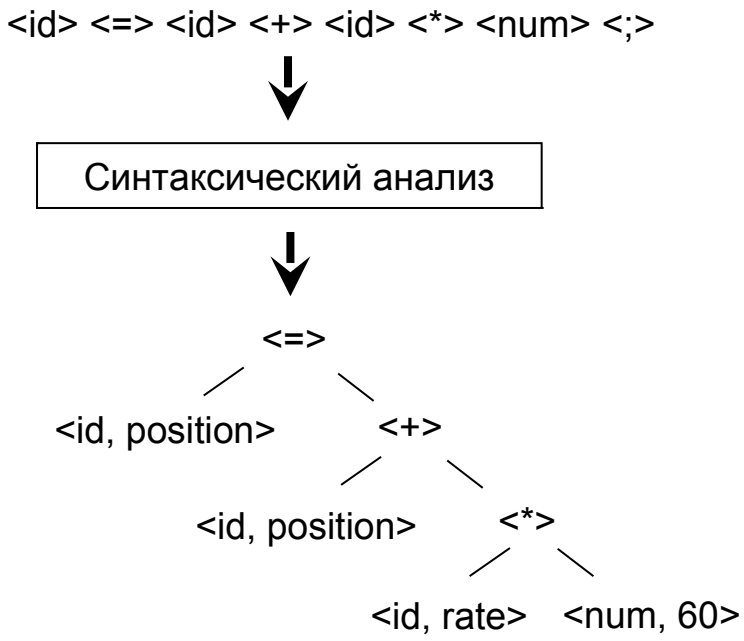

Рис. 2. Лексический анализ

В простейшем случае для простых языков на этапе синтаксического анализа могут уже производиться вычисления (например, вычисление арифметического выражения), однако чаще на этом этапе строится внутреннее представление программы, удобное для дальнейшего анализа и обработки.

Часто в качестве такого представления используется *абстрактное синтаксическое дерево (AST-дерево)*, пример которого можно видеть на Рис. 2.

Как можно видеть, AST-дерево представляет собой структурное представление исходной программы, очищенное от элементов конкретного синтаксиса (в рассматриваемом примере в AST-дерево не попал «разделитель», т.к. он не имеет отношения непосредственно к семантике данного фрагмента программы, а лишь к конкретному синтаксису языка). В качестве узлов в AST-дереве выступают операторы, к которым присоединяются их аргументы, которые в свою очередь также могут быть составными узлами. Часто узлы AST-дерева получаются из лексем, выделенных на этапе лексического анализа, однако могут встречаться и узлы, которым ни одна лексема не соответствует (на Рис. 2. AST-дерево построено полностью из лексем, кроме того показано, что узлы (как и лексемы), содержат дополнительные атрибуты – имена идентификаторов, значения чисел и т.д.).

Как правило, лексический и синтаксический анализаторы строятся (даже если построение осуществляется вручную) на основе формализованного описания лексики и синтаксиса языка. Лексика языка (правила выделения лексем) может быть описана в виде регулярных выражений или грамматик, синтаксис всегда описывается с помощью формальных грамматик (или их графического изображения в виде схем).

В общем случае грамматики состоят из правил преобразования последовательностей символов в другие последовательности. В качестве символов в грамматиках присутствуют *терминалы (терминальные символы)* и *нетерминалы (нетерминальные символы)*. Терминалами, если рассматривать грамматики для синтаксического анализа, являются классы лексем, которые выделены при разработке лексического анализатора. Нетерминалы – понятия рассматриваемого языка, например, арифметическое выражение, команда и т.д.

Грамматики бывают разных видов: порождающие/распознающие, левосторонние/правосторонние и т.д. и т.п., однако подробное изучение классификации грамматик не входит в задачу данного пособия, поэтому данные вопросы здесь не затрагиваются, а грамматики будут рассматриваться исключительно с практической стороны.

1.2. Неформальное введение в грамматики

Для описания синтаксиса формальных языков используются контекстно-свободные грамматики. Правила в таких грамматиках имеют следующий вид:

<Имя правила> → <Выражение>

Здесь «имя правила» – нетерминал, а «выражение» – своего рода шаблон для сопоставления с частью входной строки (разбираемой программы) и распознавания этой части.

Ниже приводится грамматика для разбора арифметических выражний.

```
NUMBER  ->   <число>
group   ->   "("  add  ")"
           | NUMBER
mult    ->   group  ( ( "*" | "/" )  group )*
add     ->   mult   ( ( "+" | "-" )  mult  )*
result  ->   add
```

Под <число> подразумевается, что представление чисел описано где-то за пределами этой грамматики, например в лексическом анализаторе (NUMBER записано всеми заглавными буквами, чтобы подчеркнуть, что это один из классов лексем, который должен быть определен в лексическом анализаторе). Строго говоря, все строки в кавычках также должны определяться в лексическом анализаторе, а здесь использоваться только названия выделенных классов лексем, но такая грамматика будет менее понятной и с ней неудобно будет работать.

В правой стороне от стрелочек задаются выражения грамматики («шаблоны» для распознавания), которые состоят из терминалов (фрагменты в кавычках, за которыми скрываются классы лексем) и нетерминалов (названия правил). Круглые скобки группируют часть выражения, чтобы для этих частей задать дополнительные правила применения, в данном примере это:

- <выражение>* – повторять <выражение> произвольное кол-ко раз (ноль или более);
- <выражение 1> | <выражение 2> | … | <выражение N> – применить любое из перечисленных выражений (альтернатива).

Кроме того, в грамматиках встречаются следующие правила применения:

- <выражение>+ – повторять <выражение> произвольное кол-ко раз, но не менее одного раза (один или более);
- <выражение>? – <выражение> может ка присутствовать, так и не присутствовать в разбираемой строке (опционально, ноль или один).

1.3. Нисходящий рекурсивный разбор

На Рис. 3. рассматривается разбор строки «3+4*(2+7-3)+5*10» согласно грамматике. Разбор начинается с начального правила, в рассматриваемом примере это правило result. Для нисходящего разбора применение правил можно представить в виде вызова функций, задача которой разобрать часть строки, которая соответствует шаблону, заданному в правой части правила. Каждое правило применяется (вызов «функции» осуществляется) к неразобранной части входной строки. В соответствии с грамматикой правила

могут применяться рекурсивно. В конце разбора (возврата из функции, соответствующей начальному правилу) должна оказаться разобранной вся строка.

Если часть строки окажется неразобранной, это будет означать, что входная строка не является строкой языка, описываемой грамматикой. Также строка не будет являться строкой языка, если мы не сможем применить какое-либо правило (строка не будет соответствовать шаблону в правой части правила) без наличия в грамматике других альтернатив разбора.

Таким образом, наиболее общем случаем нисходящего разбора является нисходящий разбор с возвратами. Однако на практике, как и в рассматриваемом примере, часто в процессе разбора можно выбрать однозначно выбрать подходящий вариант из альтернативы без полного их перебора (в рассматриваемой грамматике есть единственная альтернатива в правиле group и какой вариант из нее выбрать однозначно понятно, просмотрев единственный символ из неразобранной части входной строки: если это «(» – первый вариант, иначе второй.

На Рис. 3. в виде горизонтальных прямоугольников показаны обращения к правилам грамматики и прямоугольники показывают (охватывают) ту часть входной строки, которую разбирают соответствующие прямоугольникам применения правил. Если рассматривать применения правил в виде вызова функций, то на рисунке изображен стек вызовов функций: по горизонтали откладывается время (моменты) вызова функций, по вертикали – вложенные вызовы и, соответственно, глубина вложенных вызовов.

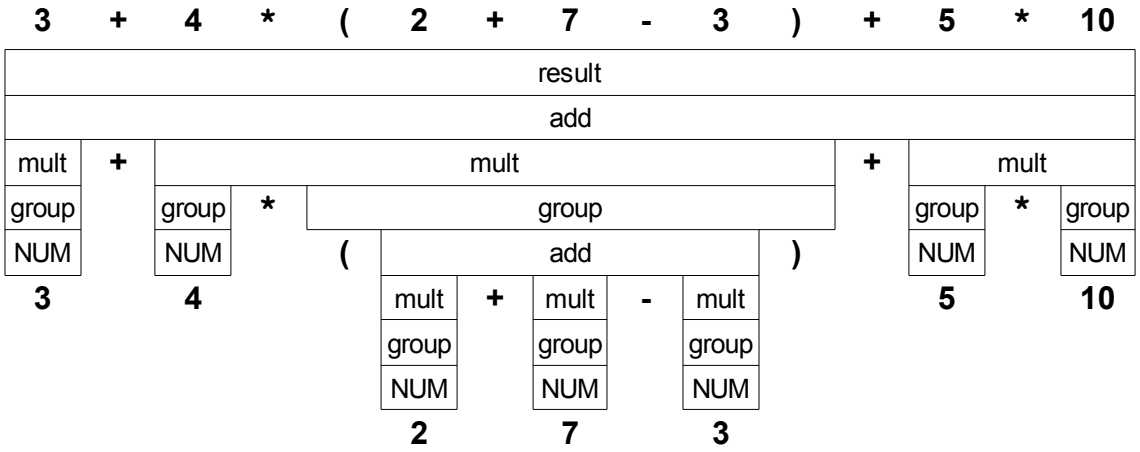

Рис. 3. Разбор строки арифметического выражения «3+4*(2+7-3)+5*10»

Собственно порядок применения правил грамматики для разбора входной строки будет называть *деревом разбора* (можно встретить также название *конкретное синтаксическое дерево* по аналогии с абстрактным). Дерево разбора для рассматриваемого примера приведено на Рис. 4., как можно видеть, оно полностью соответствует Рис. 3., только в несколько в другом

представлении.

Нетрудно догадаться, что синтаксический анализатор, осуществляющий нисходящий рекурсивный разбор, можно реализовать вручную (единственный метод разбора, который поддается реализации вручную), если описывать каждое правило грамматики в виде функции, тело которой будет соответствовать правой части правила, причем перевод выражений грамматики в код функции будет достаточно формальной процедурой.

На Рис. 5. приведено AST-дерево, которое соответствует арифметическому выражению «3+4*(2+7-3)+5*10». Очевидно, что дальнейший анализ и обработка (например, вычисление выражения) гораздо удобнее осуществлять над AST-деревом, чем над деревом разбора. Кроме того, даже при эквивалентных изменениях грамматики (описанный грамматикой язык не меняется) дерево разбора будет меняться, что категорически неудобно.

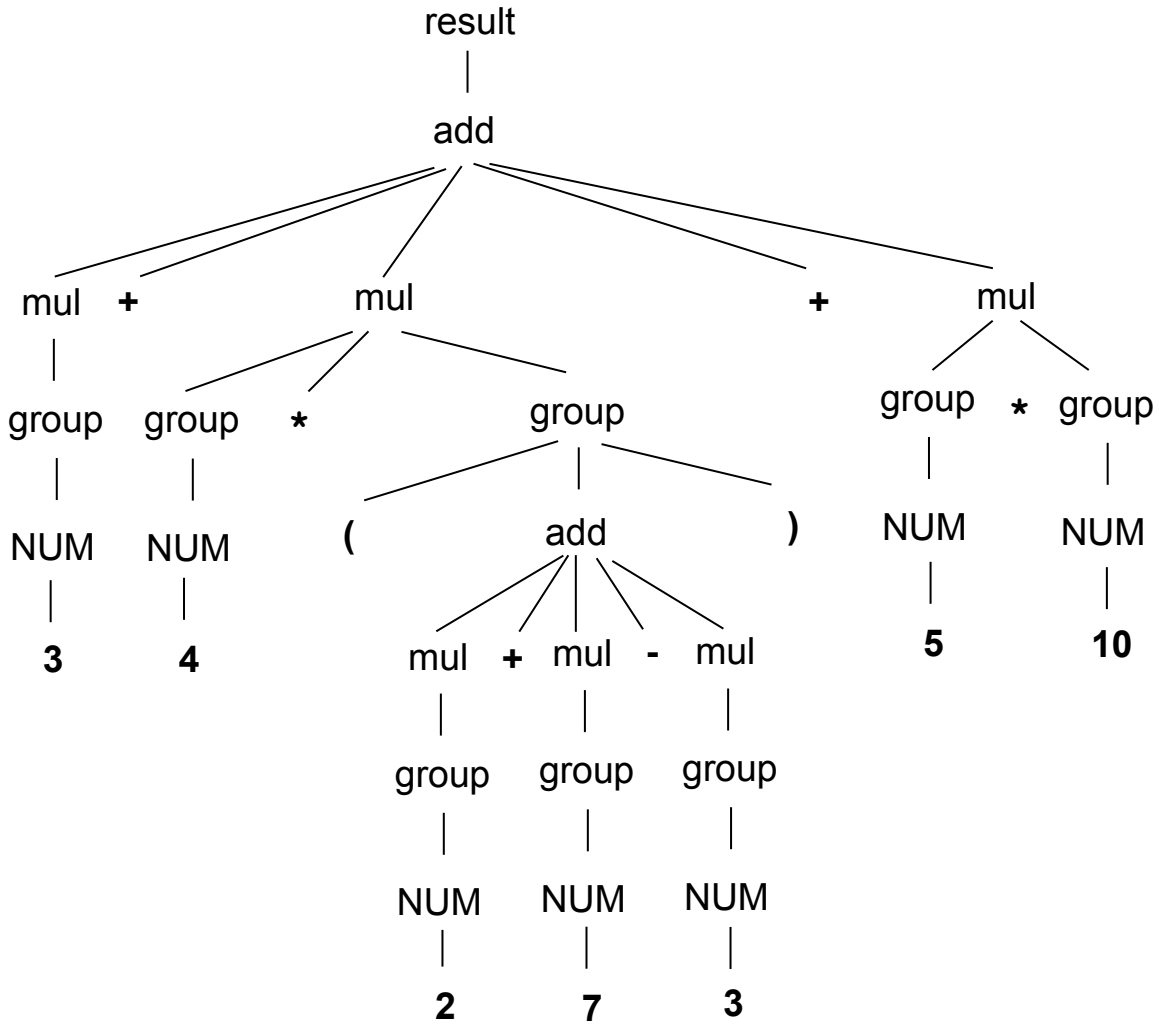

Рис. 4. Дерево разбора арифметического выражения «3+4*(2+7-3)+5*10»

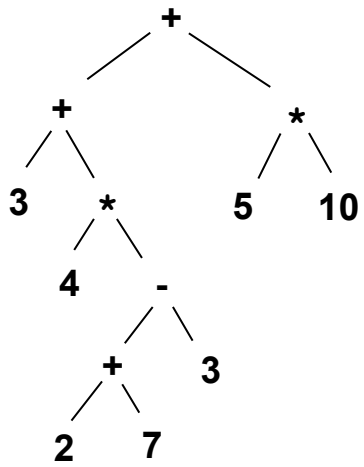

Рис. 5. AST-дерево арифметического выражения «3+4*(2+7-3)+5*10»

2. Построение синтаксических анализаторов без использования специализированных инструментальных средств («вручную»)

Как уже говорилось выше, синтаксический анализатор, реализующий нисходящий рекурсивный разбор, можно получить из подходящей грамматики (как минимум, не является леворекурсивной) достаточно формально.

Но надо строго придерживаться правила, что разработку парсера необходимо начинать именно с составления грамматики, а код писать по уже составленной грамматике. Если в синтаксис вводятся изменения, то сначала модифицировать грамматику, а только потом исправлять код. В противном случае вероятность запутаться очень велика, т.к. понимать и описывать синтаксис языков в виде грамматик гораздо легче, чем сразу в коде (очевидно, что формализм грамматик был предложен не просто так :-).

В данном разделе рассматривается пример построения парсера (и вычислителя в одном лице) для арифметических выражений, соответствующего грамматике, которая рассматривалась выше.

Стоит сразу уточнить, что при реализации синтаксических анализаторов вручную очень часто функциональность лексического и синтаксического анализаторов явно не разделяется и реализуется вместе, что и было сделано в первом рассматриваемом примере.

2.1. Класс ParserBase – базовый класс для реализации парсера

Прежде чем непосредственно рассматривать приемы перевода выражений грамматики в код функций разбора, следует познакомиться с вспомогательным классом, который обеспечивает базовые функции в работе любого парсера. Код данного класса с комментариями приводится ниже.

```
namespace MathExpr
{
  public class ParserBase
  {
    // незначащие символы - пробельные символы по умолчанию
    public const string DEFAULT_WHITESPACES = " \n\r\t";

    // разбираемая строка
    private string source = null;
    // позиция указателя
    // (указывает на первый символ неразобранной части вход. строки)
    private int pos = 0;

    public ParserBase(string source) {
      this.source = source;
    }
```

```
public string Source {
  get { return source; }
}

public int Pos {
  get { return pos; }
}

// предотвращает возникновение ошибки обращения за пределы
// массива; в этом случае возвращает (char) 0,
// что означает конец входной строки
protected char this[int index] {
  get {
    return index < source.Length ? source[index] : (char) 0;
  }
}

// символ в текущей позиции указателя
public char Current {
  get { return this[Pos]; }
}

// определяет, достигнут ли конец строки
public bool End {
  get {
    return Current == 0;
  }
}

// передвигает указатель на один символ
public void Next() {
  if (!End)
    pos++;
}

// пропускает незначащие (пробельные) символы
public virtual void Skip() {
  while (DEFAULT_WHITESPACES.IndexOf(this[pos])>=0)
    Next();
}

// распознает одну из строк; при этом указатель смещается и
// пропускаются незначащие символы;
// если ни одну из строк распознать нельзя, то возвращается null
protected string MatchNoExcept(params string[] terms) {
  int pos = Pos;
  foreach (string s in terms) {
    bool match = true;
```

```
      foreach (char c in s)
        if (Current == c)
          Next();
        else {
          this.pos = pos;
          match = false;
          break;
        }
      if (match) {
        // после разбора терминала пропускаем незначащие символы
        Skip();
        return s;
      }
    }
    return null;
  }

  // проверяет, можно ли в текущей позиции указателя, распознать
  // одну из строк; указатель не смещается
  public bool IsMatch(params string[] terms) {
    int pos = Pos;
    string result = MatchNoExcept(terms);
    this.pos = pos;
    return result != null;
  }


  // распознает одну из строк; при этом указатель смещается и
  // пропускаются незначащие символы; если ни одну из строк
  // распознать нельзя, то выбрасывается исключение
  public string Match(params string[] terms) {
    int pos = Pos;
    string result = MatchNoExcept(terms);
    if (result == null) {
      string message = "Ожидалась одна из строк: ";
      bool first = true;
      foreach (string s in terms) {
        if (!first)
          message += ", ";
        message += string.Format("\"{0}\"", s);
        first = false;
      }
      throw new ParserBaseException(
                  string.Format("{0} (pos={1})", message, pos));
    }
    return result;
  }

  // то же, что и Match(params string[] a), для удобства
```

```
    public string Match(string s) {
      int pos = Pos;
      try {
        return Match(new string[] { s });
      }
      catch {
        throw new ParserBaseException(
                    string.Format(
                      "{0}: '{1}' (pos={2})",
                      s.Length == 1 ? "Ожидался символ"
                                    : "Ожидалась строка",
                      s, pos
                    )
                );
      }
    }
  }
}
```

Основная задача приведенного выше класса ParserBase по возможности избавить разработчика парсера от необходимости работать на уровне символов входной строки, позиции указателя и т. п.

2.2. Перевод правил грамматики в код функций парсера

Ниже приводится код «вычислителя» правильных арифметических выражений, заданных в виде строки. Данный класс для вычислений разбирает арифметическое выражение, т.е. является, в том числе, полноценным синтаксическим анализатором.

```
using System.Globalization;


namespace MathLang
{
  public class MathExprIntepreter: ParserBase
  {
    // "культуронезависимый" формат для чисел (с разделителем ".")
    public static readonly NumberFormatInfo NFI =
      new NumberFormatInfo();

    // конструктор
    public MathExprIntepreter(string source) : base(source) {
    }

    // далее идет реализация в виде функций правил грамматики
```

```
//  NUMBER ->  <число>  (реализация в грамматике не описана)
public double NUMBER() {
  string number = "";
  while (Current == '.' || char.IsDigit(Current)) {
    number += Current;
    Next();
  }
  if (number.Length == 0)
    throw new ParserBaseException(
      string.Format("Ожидалось число (pos={0})", Pos));
  Skip();

  return double.Parse(number, NFI);
}

//  group ->  "("  add  ")"  |  NUMBER
public double Group() {
  if (IsMatch("(")) {  // выбираем альтернативу
    Match("(");          // это выражение в скобках
    double result = Add();
    Match(")");
    return result;
  }
  else
    return NUMBER();     // это число
}

//  mult ->  group  ( ( "*" | "/" )  group )*
public double Mult() {
  double result = Group();
  while (IsMatch("*", "/")) {  // повторяем нужное кол-во раз
    string oper = Match("*", "/");  // здесь выбор альтернативы
    double temp = Group();          // реализован иначе
    result = oper == "*" ? result * temp
                         : result / temp;
  }
  return result;
}

//  add ->  mult   ( ( "+" | "-" )  mult  )*
public double Add() {  //  реализация аналогично правилу mult
  double result = Mult();
  while (IsMatch("+", "-")) {
    string oper = Match("+", "-");
    double temp = Mult();
    result = oper == "+" ? result + temp
                         : result - temp;
  }
  return result;
```

```
    }

    //  result ->  add
    public double Result() {
      return Add();
    }

    // метод, вызывающий начальное и правило грамматики и
    // соответствующие вычисления
    public double Execute() {
      Skip();
      double result = Result();
      if (End)
        return result;
      else
        throw new ParserBaseException(  // разобрали не всю строку
          string.Format("Лишний символ '{0}' (pos={1})",
                        Current, Pos)
        );
    }

    // статическая реализации предыдузего метода (для удобства)
    public static double Execute(string source) {
      MathExprIntepreter mei = new MathExprIntepreter(source);
      return mei.Execute();
    }
  }
}
```

Для вычисления арифметических выражений, представленных в виде строки, достаточно вызвать метод Execute класса MathExprIntepreter:

```
  double result = MathExprIntepreter.Execute("3+4*(2+7-3)+5*10");
```

Как уже говорилось и можно видеть из листинга перевод выражений грамматики в код достаточно формальная процедура:

- Терминальные символы заменяются вызовом Match("<строка>").
- Нетеминальные символы (правила) – вызовом соответствующей процедуры.
- Повторения (обозначаются * и +) – циклами while или do … while. В качестве условия цикла чаще всего выступает вызов функции IsMatch(...) или поверка окончание разбора всей строки через свойство End
- Опциональное выражение (обозначается ?) – условным оператором. В качестве условия чаще всего выступает вызов функции

IsMatch(...).

Естественно, помимо непосредственно разбора строки (сопоставления грамматике) выполняются необходимые действия. В примере выше производится непосредственно вычисление арифметического выражения.

2.3. Построение AST-дерева в процессе разбора

Однако гораздо чаще, как уже говорилось выше, на этапе синтаксического анализа необходимо построить AST-дерево программы, а дальнейшую обработку (интерпретация, компиляция и т.п.) выполнять уже над AST-деревом. Ниже рассматривается доработка рассматриваемого примера до простейшего языка вычислений, в котором поддерживаются глобальные переменные, арифметические операции, операция присвоения, а также ввода/вывод данных. При этом на этапе синтаксического анализа происходит именно построение AST-дерева, а вычисления (интерпретация) реализованы над AST-деревом в другом классе.

Как уже подчеркивалось, доработку первоначального примера надо начинать с доработки грамматики. Для нашего простейшего языка грамматика будет иметь следующий вид:

```
NUMBER  ->   <число>
IDENT   ->   <идентификатор>
group   ->   "("  term  ")"
           | IDENT
           | NUMBER
mult    ->   group  ( ( "*" | "/" )  group )*
add     ->   mult   ( ( "+" | "-" )  mult  )*
term    ->   add
expr    ->   "print"  term
           | "input"  IDENT
           | IDENT  "="  term
program ->   (  Expr  )*
result  ->   program
```

Прежде, чем рассмотреть код синтаксического анализатора, который соответствует это грамматике и строит AST-дерево, следует познакомиться с классом, который используется для создания узлов AST-дерева.

```
using System.Collections.Generic;

namespace MathLang
{
  public class AstNode
  {
    // тип узла (см. описание ниже)
```

```
    public virtual int Type { get; set; }
    // текст, связанный с узлом
    public virtual string Text { get; set; }

    // родительский узел для данного узла дерева
    private AstNode parent = null;
    // потомки (ветви) данного узла дерева
    private IList<AstNode> childs = new List<AstNode>();

    // конструкторы с различными параметрами (для удобства
    public AstNode(int type, string text,
                   AstNode child1, AstNode child2) {
      Type = type;
      Text = text;
      if(child1 != null)
        AddChild(child1);
      if(child2 != null)
        AddChild(child2);
    }
    public AstNode(int type, AstNode child1, AstNode child2)
      : this(type, null, child1, child2) {
    }
    public AstNode(int type, AstNode child1)
      : this(type, child1, null) {
    }
    public AstNode(int type, string label)
      : this(type, label, null, null) {
    }
    public AstNode(int type)
      : this(type, (string) null) {
    }

    // метод добавления дочернего узла
    public void AddChild(AstNode child) {
      if (child.Parent != null) {
        child.Parent.childs.Remove(child);
      }
      childs.Remove(child);
      childs.Add(child);
      child.parent = this;
    }

    // метод удаления дочернего узла
    public void RemoveChild(AstNode child) {
      childs.Remove(child);
      if (child.parent == this)
        child.parent = null;
    }
```

```
    // метод получения дочернего узла по индексу
    public AstNode GetChild(int index) {
      return childs[index];
    }

    // метод добавления дочернего узла
    public int ChildCount {
      get {
        return childs.Count;
      }
    }

    // родительский узел (свойство)
    public AstNode Parent {
      get {
        return parent;
      }
      set {
        value.AddChild(this);
      }
    }

    // индекс данного узла в дочерних узлах родительского узла
    public int Index {
      get {
        return Parent == null ? -1
                              : Parent.childs.IndexOf(this);
      }
    }

    // представление узла в виде строки
    public override string ToString() {
      return Text != null ? Text
                          : AstNodeType.AstNodeTypeToString(Type);
    }
  }

  // класс констант для перечисления возможных типов токенов
  public class AstNodeType
  {
    public const int UNKNOWN = 0;

    public const int NUMBER  = 1;
    public const int IDENT   = 5;

    public const int ADD     = 11;
    public const int SUB     = 12;
    public const int MUL     = 13;
    public const int DIV     = 14;
```

```
    public const int ASSIGN  = 51;
    public const int INPUT   = 52;
    public const int PRINT   = 53;

    public const int BLOCK   = 100;
    public const int PROGRAM = 101;


    public static string AstNodeTypeToString(int type) {
      switch(type) {
        case UNKNOWN: return "?";
        case NUMBER:  return "NUM";
        case IDENT:   return "ID";
        case ADD:     return "+";
        case SUB:     return "-";
        case MUL:     return "*";
        case DIV:     return "/";
        case ASSIGN:  return "=";
        case INPUT:   return "input";
        case PRINT:   return "print";
        case BLOCK:   return "..";
        case PROGRAM: return "program";
        default:      return "";
      }
    }
  }
}
```

Реализация кода парсера простейшего языка арифметических вычислений с использование класса AstNode представлена ниже:

```
namespace MathLang
{
  public class MathLangParser: ParserBase
  {
    // конструктор
    public MathLangParser(string source)
      : base(source) {
    }

    // далее идет реализация в виде функций правил грамматики

    //  NUMBER ->  <число>
    public AstNode NUMBER() {
      string number = "";
      while (Current == '.' || char.IsDigit(Current)) {
        number += Current;
```

```
      Next();
    }
    if (number.Length == 0)
      throw new ParserBaseException(
        string.Format("Ожидалось число (pos={0})", Pos));
    Skip();

    return new AstNode(AstNodeType.NUMBER, number);
  }

  //  IDENT ->  <идентификатор>
  public AstNode IDENT() {
    string identifier = "";
    if (char.IsLetter(Current)) {
      identifier += Current;
      Next();
      while (char.IsLetterOrDigit(Current)) {
        identifier += Current;
        Next();
      }
    }
    else
      throw new ParserBaseException(
        string.Format("Ожидался идентификатор (pos={0})", Pos));
    Skip();

    return new AstNode(AstNodeType.IDENT, identifier);
  }

  //  group ->  "("  term  ")"  |  IDENT |  NUMBER
  public AstNode Group() {
    if (IsMatch("(")) {  // выбираем альтернативу
      Match("(");          // это выражение в скобках
      AstNode result = Term();
      Match(")");
      return result;
    }
    else if (char.IsLetter(Current)) {
      int pos = Pos;       // это идентификатор
      return IDENT();
    }
    else
      return NUMBER();     // число
  }

  //  mult ->  group  ( ( "*" | "/" )  group )*
  public AstNode Mult() {
    AstNode result = Group();
    while (IsMatch("*", "/")) {  // повторяем нужное кол-во раз
```

```
      string oper = Match("*", "/");  // здесь выбор альтернативы
      AstNode temp = Group();         // реализован иначе
      result =
        oper == "*" ? new AstNode(AstNodeType.MUL, result, temp)
                    : new AstNode(AstNodeType.DIV, result, temp);
    }
    return result;
  }

  //  add ->  mult  ( ( "+" | "-" )  mult  )*
  public AstNode Add() {  //  реализация аналогично правилу mult
    AstNode result = Mult();
    while (IsMatch("+", "-")) {
      string oper = Match("+", "-");
      AstNode temp = Mult();
      result =
        oper == "+" ? new AstNode(AstNodeType.ADD, result, temp)
                    : new AstNode(AstNodeType.SUB, result, temp);
    }
    return result;
  }

  //  term ->  add
  public AstNode Term() {
    return Add();
  }

  //  expr ->  "print" term  |  "input" IDENT  |  IDENT "=" term
  public AstNode Expr() {
    if (IsMatch("print")) {  // выбираем альтернативу
      Match("print");           // это вывод данных
      AstNode value = Term();
      return new AstNode(AstNodeType.PRINT, value);
    }
    else if (IsMatch("input")) {
      Match("input");           // это ввод данных
      AstNode identifier = IDENT();
      return new AstNode(AstNodeType.INPUT, identifier);
    }
    else {
      AstNode identifier = IDENT();
      Match("=");               // это операция присвоения значения
      AstNode value = Term();
      return new AstNode(AstNodeType.ASSIGN, identifier, value);
    }
  }

  //  program ->  (  expr  )*
  public AstNode Program() {
```

```
      AstNode programNode = new AstNode(AstNodeType.PROGRAM);
      while (!End)  // повторяем до конца входной строки
        programNode.AddChild(Expr());
      return programNode;
    }

    //  result ->  program
    public AstNode Result() {
      return Program();
    }

    // метод, вызывающий начальное и правило грамматики и
    // соответствующий парсинг
    public AstNode Parse() {
      Skip();
      AstNode result = Result();
      if (End)
        return result;
      else
        throw new ParserBaseException(  // разобрали не всю строку
          string.Format("Лишний символ '{0}' (pos={1})",
                        Current, Pos)
        );
    }

    // статическая реализации предыдузего метода (для удобства)
    public static AstNode Parse(string source) {
      MathLangParser mlp = new MathLangParser(source);
      return mlp.Parse();
    }
  }
}
```

2.4. Вывод на экран построенных AST-деревьев

Ниже приводится класс для печати в стандартный поток вывода (на консоль) построенного AST-дерева с помощью псевдографики (в кодировке CP866 – DOS). Данный класс удобно использовать для отладки (код данного в том числе совместим с классами runtime-библиотеки ANTLT 3 для языка C#).

```
using System;
using System.Text;


namespace MathLang
{
  public class AstNodePrinter
```

```
{
  public const byte ConnectCharDosCode    = 0xB3,
                    MiddleNodeCharDosCode = 0xC3,
                    LastNodeCharDosCode   = 0xC0;

  public static readonly char ConnectChar    = '|',
                              MiddleNodeChar = '*',
                              LastNodeChar   = '-';

  static AstNodePrinter() {
    Encoding dosEncoding = null;
    try {
      dosEncoding = Encoding.GetEncoding("cp866");
    }
    catch { }
    if (dosEncoding != null) {
      ConnectChar = dosEncoding.GetChars(
                          new byte[] { ConnectCharDosCode })[0];
      MiddleNodeChar = dosEncoding.GetChars(
                          new byte[] { MiddleNodeCharDosCode })[0];
      LastNodeChar = dosEncoding.GetChars(
                          new byte[] { LastNodeCharDosCode })[0];
    }
  }

  private static string getStringSubTree(AstNode node,
                           string indent, bool root) {
    if (node == null)
      return "";

    string result = indent;
    if (!root)
      if(node.Index < node.Parent.ChildCount - 1) {
        result += MiddleNodeChar + " ";
        indent += ConnectChar + " ";
      }
      else {
        result += LastNodeChar + " ";
        indent += "  ";
      }
    result += node + "\n";
    for(int i = 0; i < node.ChildCount; i++)
      result += getStringSubTree(node.GetChild(i), indent, false);

    return result;
  }

  public static string astNodeToAdvancedDosStringTree(
                           AstNode node) {
```

```
      return getStringSubTree(node, "", true);
    }

    public static void Print(AstNode node) {
      string tree = astNodeToAdvancedDosStringTree(node);
      Console.WriteLine(tree);
    }
  }
}
```

2.5. Выполнение вычислений над AST-деревом

Несмотря на то, что интерпретация AST-дерева не относится к синтаксическому анализу, ниже приводится код интерпретатора рассматриваемого языка:

```
using System;
using System.Collections.Generic;
using System.Globalization;


namespace MathLang
{
  public class MathLangIntepreter
  {
    // "культуронезависимый" формат для чисел (с разделителем ".")
    public static readonly NumberFormatInfo NFI =
      new NumberFormatInfo();

    // таблица переменных
    private Dictionary<string, double> varTable =
      new Dictionary<string, double>();

    // корневой узел AST-дерева программы
    private AstNode programNode = null;

    // конструктор
    public MathLangIntepreter(AstNode programNode) {
      if (programNode.Type != AstNodeType.PROGRAM)
        throw new IntepreterException(
                    "AST-дерево не является программой");
      this.programNode = programNode;
    }

    // рекурсивный метод, который вызывается для каждого узла дерева
    private double ExecuteNode(AstNode node) {
      switch (node.Type) {
```

```
        case AstNodeType.UNKNOWN:
          throw new IntepreterException(
                      "Неопределенный тип узла AST-дерева");

        case AstNodeType.NUMBER:
          return double.Parse(node.Text, NFI);

        case AstNodeType.IDENT:
          if (varTable.ContainsKey(node.Text))
            return varTable[node.Text];
          else
            throw new ParserBaseException(string.Format(
                          "Значение {0} не определено", node.Text));

        case AstNodeType.ADD:
          return ExecuteNode(node.GetChild(0)) +
                   ExecuteNode(node.GetChild(1));

        case AstNodeType.SUB:
          return ExecuteNode(node.GetChild(0)) -
                   ExecuteNode(node.GetChild(1));

        case AstNodeType.MUL:
          return ExecuteNode(node.GetChild(0)) *
                   ExecuteNode(node.GetChild(1));

        case AstNodeType.DIV:
          return ExecuteNode(node.GetChild(0)) /
                   ExecuteNode(node.GetChild(1));

        case AstNodeType.ASSIGN:
          varTable[node.GetChild(0).Text] =
            ExecuteNode(node.GetChild(1));
          break;

        case AstNodeType.INPUT:
          Console.Write("input {0}: ", node.GetChild(0).Text);
          varTable[node.GetChild(0).Text] =
            double.Parse(Console.ReadLine(), NFI);
          break;

        case AstNodeType.PRINT:
          Console.WriteLine(
            ExecuteNode(node.GetChild(0)).ToString(NFI));
          break;

        case AstNodeType.BLOCK:
        case AstNodeType.PROGRAM:
          for (int i = 0; i < node.ChildCount; i++)
```

```
            ExecuteNode(node.GetChild(i));
          break;

        default:
          throw new IntepreterException(
                      "Неизвестный тип узла AST-дерева");
      }

      return 0;
    }

    // public-метод для вызова интерпретации
    public void Execute() {
      ExecuteNode(programNode);
    }

    // статическая реализации предыдузего метода (для удобства)
    public static void Execute(AstNode programNode) {
      MathLangIntepreter mei=new MathLangIntepreter(programNode);
      mei.Execute();
    }
  }
}
```

2.6. Сборка проекта, тестирование, промежуточные итоги

Для запуска проекта необходима следующая реализация метода Main:

```
using System;
using System.IO;


namespace MathLang
{
  public class Program
  {
    static void Main(string[] args) {
      // в зависимости от наличия параметров командной строки
      // разбираем либо файл с именем, переданным первым параметром
      // командной строки, либо стандартный ввод
      TextReader reader =
        args.Length >= 1 ? new StreamReader(args[0])
                         : Console.In;
      String source = reader.ReadToEnd();
      try {
        AstNode program = MathLangParser.Parse(source);
        AstNodePrinter.Print(program);
        Console.WriteLine("------------------------");
```

```
        MathLangIntepreter.Execute(program);
      }
      catch (Exception e) {
        Console.WriteLine("Error: {0}", e);
      }
      Console.ReadLine();
    }
  }
}
```

Рассмотренный выше проект может выполнить, например, следующий код (файл input.txt):

```
print 0
A = 7  input B

C         =A
+B*B

print C / 2
```

Форматирование приводится намеренно такое «нечитаемое», чтобы показать, что полученный синтаксический анализатор вполне с ним справится. В результате выполнения команды «MathLang.exe input.txt» и вводе для переменной B значения 10 распечатается следующий текст (вначале AST-дерево, затем результаты интерпретации):

```
program
├ print
│ └ 0
├ =
│ ├ A
│ └ 7
├ input
│ └ B
├ =
│ ├ C
│ └ +
│   ├ A
│   └ *
│     ├ B
│     └ B
└ print
  └ /
    ├ C
    └ 2
```

```
-----------------------
0
input B: 10
53.5
```

В данном разделе были рассмотрены два проекта. Первый проект – вычисление арифметического выражения, на этапе разбора осуществляется вычисление арифметического выражения. Второй проект – доработка первого до простейшего языка, на этапе разбора осуществляется построение AST-дерева программы (внутренне представление программы); выполнение программы осуществляется уже над AST-деревом после разбора.

Довольно просто и тривиально доработать второй проект до простейшего интерпретируемого языка программирования с циклами, условным оператором и т.д. Здесь же хотелось подчеркнуть следующую мысль.

В случае необходимости реализации синтаксического анализатора достаточно сложного формального языка следует осуществлять разработку снизу вверх, начиная от элементарных выражений к все более сложным синтаксическим конструкциям, с тестированием промежуточного результата на каждом шаге. Так дорабатывать последний рассмотренный проект можно было бы следующим образом:

- добавить составной оператор;
- добавить операторы сравнения и логические операторы;
- добавить условный оператор;
- добавить циклы;
- добавить описание и вызов функций;
- и т.д.

При этом на каждом шаге получался бы работоспособный парсер, который мог бы разбирать какое-то подмножество требуемого формального языка.

3. Построение синтаксических анализаторов с использованием ANTLR 3

Выше рассматривались приемы построения синтаксических анализаторов вручную. Однако процесс построения при этом для более сложных формальных языков в большинстве случаев, не то, чтобы очень сложен, но достаточно трудоемок.

Существуют достаточно большое количество специализированных инструментов, позволяющих процесс построения синтаксических анализаторов упростить и автоматизировать. В большинстве случаев предпочтительнее воспользоваться одним их тих инструментов (построение синтаксических анализаторов вручную выше рассматривалось в большей степени в методических целях).

Широко известны следующие инструменты для построения синтаксических анализаторов:

- Yacc / Bizon (совместно с Flex – для построения лексических анализаторов)
- Coco/R
- JavaCC
- ANTLR

Ниже рассматриваются приемы построения парсеров с использованием ANTLR 3. ANTLR выбран в качестве примера как одно из наиболее известных и широко используемых средств автоматизации построения синтаксических анализаторов для различных языков программирования.

В настоящий момент уже существует ANTLR v4.4, однако в методическом плане он менее подходящий, прежде всего потому, что в ANTLR 4 по непонятной причине прекращена поддержка крайне удобного механизма автоматического построения AST-деревьев, которая присутствовала в ANTLR 2.7 и 3.

3.1. Знакомство с ANTLR 3

ANTLR (от ANother Tool for Language Recognition – «Еще один Инструмент для Распознавания Языков») – это инструментарий для создания парсеров формальных языков, которые в свою очередь могут использоваться в компиляторах или интерпретаторах языков программирования или DSL (Domain Specific Language – язык предметной области), инструментах анализа кода (например, статических анализаторов кода) и других языковых инструментах.

Страницей проекта ANTLR в интернете является http://www.antlr.org/,

информация о ANTLR 3 доступна по адресу http://www.antlr3.org/.

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. Неполный список проектов, в которых был использован ANTLR, можно посмотреть по адресу http://www.antlr3.org/showcase/list.html (среди них Groovy, Hibernate, Twitter's search query language, IntelliJ IDEA и др.)

ANTLR состоит из 2-х основных 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 и др. Для всех этих языков (называемых «целевыми» – target languages) имеются портированные версии runtime-библиотеки. В данном пособии рассматривается применение ANTLR для проектов на языке C#.

В дополнение к перечисленным инструментом, для ANTLR существует специализированная среда разработки ANTLTWorks, которая позволяет редактировать (с подсветкой синтаксиса и подстановки кода) и отлаживать грамматики.

3.2. Принцип работы ANTLR 3

ANTLR 3 умеет не только строить лексические и синтаксические анализаторы, но и содержит средства для работы с построенными AST-деревьями. Для этого ANTLR поддерживает так называемые древовидные

грамматики (Tree-grammars), с помощью которых можно реализовать модули для обхода построенных AST-деревьев и выполнения необходимых действий (семантический анализ, генерация кода и т.д.). Однако в данном методическом пособии данные возможности ANTLR рассматриваться не будут, т.к. непосредственно к синтаксическому анализу не относятся (кроме того, методически более правильным для студентов будет работа с построенными деревьями «вручную», что не представляет какой-либо сложности, см. пример класса MathLangIntepreter из предыдущего раздела).

Общий принцип работы с ANTLR приведен на Рис. 6.

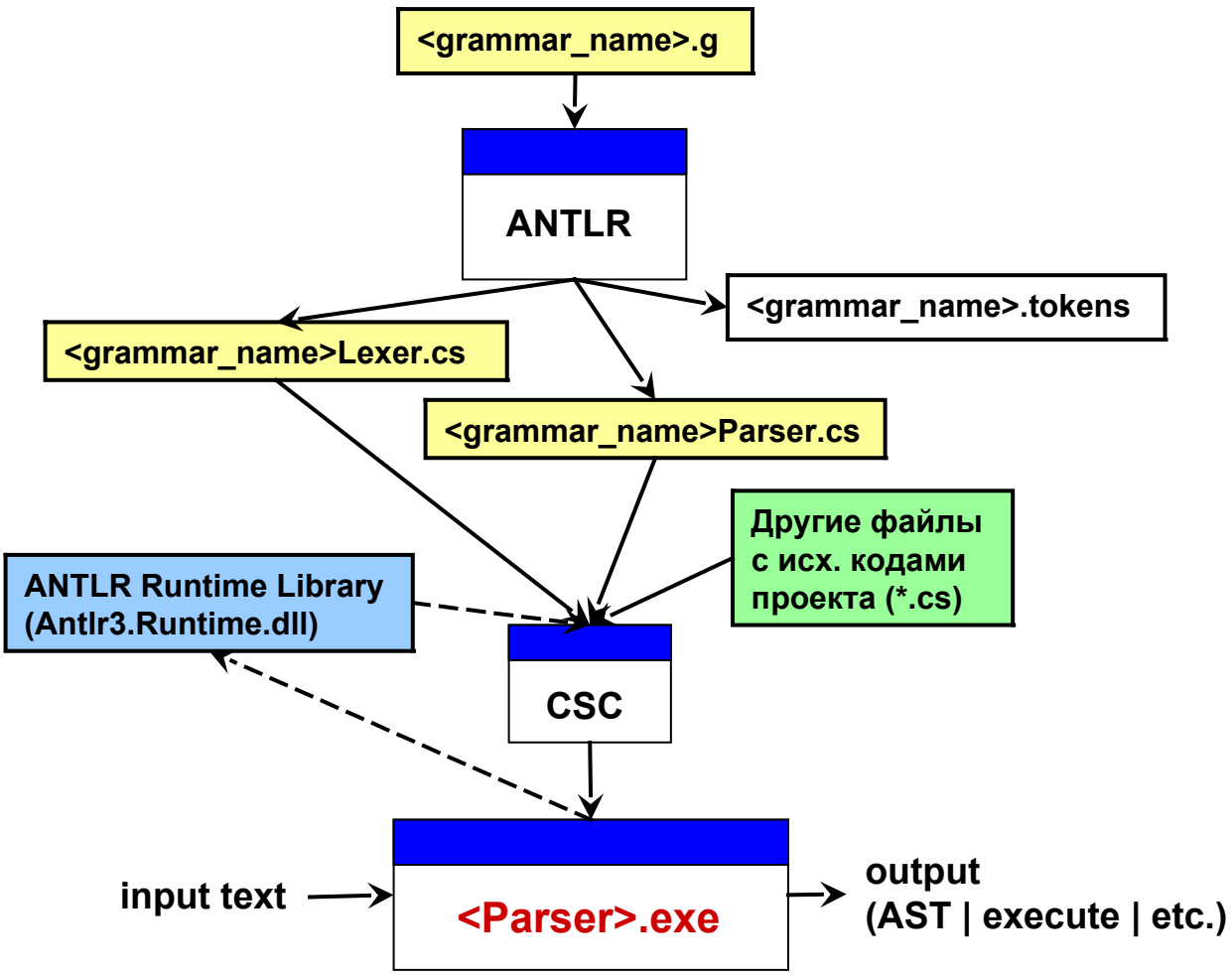

Рис. 6. Построение парсера с помощью ANTLR (пример для C#)

Как видно из Рис. 6. Основной задачей ANTLR является построение лексического и синтаксического анализатора из файла грамматики (расширение *.g). Таким образом программист избавляется от необходимости ручного перевода грамматики в код синтаксического анализатора (данный вопрос рассматривался в предыдущем разделе).

ANTLR по файлу грамматики <grammar_name>.g генерирует три файла <grammar_name>Lexer.cs, в котором описан класс лексического анализатора,

<grammar_name>Parser.cs, в котором описан класс синтаксического анализатора, а также текстовый файл <grammar_name>.tokens, в котором перечислены все типы токенов, которые ANTLR выделил при построении лексического и синтаксического анализаторов.

Полученные файлы (<grammar_name>Lexer.cs, <grammar_name>Parser.cs) вместе с остальными исходными кодами проекта компилируются в исполняемую программу (или динамическую библиотеку) с помощью компилятора языка C#. Т.к. сгенерированный код лексического и синтаксического анализатора использует классы из Runtime-библиотеки (в данном случае для языка C# – Antlr3.Runtime.dll), то при компиляции необходимо указать ссылку на сборку с Runtime-библиотекой.

Полученная исполняемая программа (компилятор, интерпретатор и т.д.) также будет требовать наличие сборки с Runtime-библиотекой (при желании Runtime-библиотеку в виде исходных кодов можно статически скомпилировать с проектом, в этом случае сборка Antlr3.Runtime.dll для выполнения скомпилированной программы не будет нужна).

Важным моментов является использование ANTLR и Runtime-библиотеки одной и той же или совместимых версий, иначе либо проект невозможно будет скомпилировать, либо во время работы конечной программы в процессе синтаксического анализа возможны ошибки.

Т.к. утилита ANTLR разработана на языке Java, то для ее запуска необходимо для генерации лексического и синтаксического анализаторов необходимо выполнить следующую команду (версии ANTLR может отличаться):

```
java -jar antlr-3.3-complete.jar <grammar_name>.g
```

В конце данного пособия рассматривается прием, позволяющий добавить данную команду в процесс сборки проекта Visual Studio, чтобы сборка осуществлялась «одной кнопкой».

Во время разбора взаимодействие составных частей полученного парсера показано на Рис. 7. Как видно из Рис. 7., в парсерах, построенных с помощью ANTLR, фазы лексического и синтаксического анализа разделены (ANTLR отвечает за построение и лексического и синтаксического анализаторов. Лексический анализатор отвечает за разбиение потока символов входной строки на поток лексем, также называемых токенами. Синтаксический анализатор чаще всего строит AST-дерево из потока токенов. Как уже говорилось выше, ANTLR в том числе содержит средства (которые в данном пособии не рассматриваются) для построения обработчиков полученных AST-деревьев, однако AST-деревья достаточно просто обрабатывать и «вручную».

На Рис. 8. для наглядности показана последовательность преобразования потока символов входной строки сначала в поток токенов, а затем в AST-дерево. Следует обратить внимание, что не все токены попадают в AST-дерево Не попали токены «пробелы» (WS) и токен «точка с запятой», однако причины

отсутствия этих токенов в AST-дереве различны. Токены «пробелы» (WC) в потоке токенов помечены зачеркнутым углом, что означает, что данные токены полностью игнорируются синтаксическим анализатором. Токен «точка с запятой» в синтаксическом анализаторе учитывает при разборе (важен в «конкретном» синтаксисе языка), однако при построении AST-дерева пропускает как незначащий (для «абстрактного» синтаксиса языка).

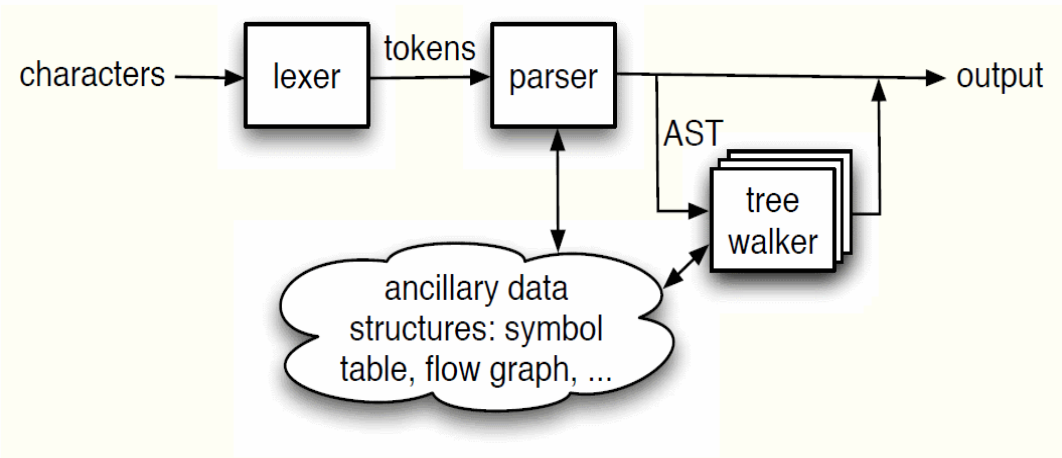

Рис. 7. Общая архитектура и взаимодействие во время разбора составных частей построенных с помощью ANTLR парсеров

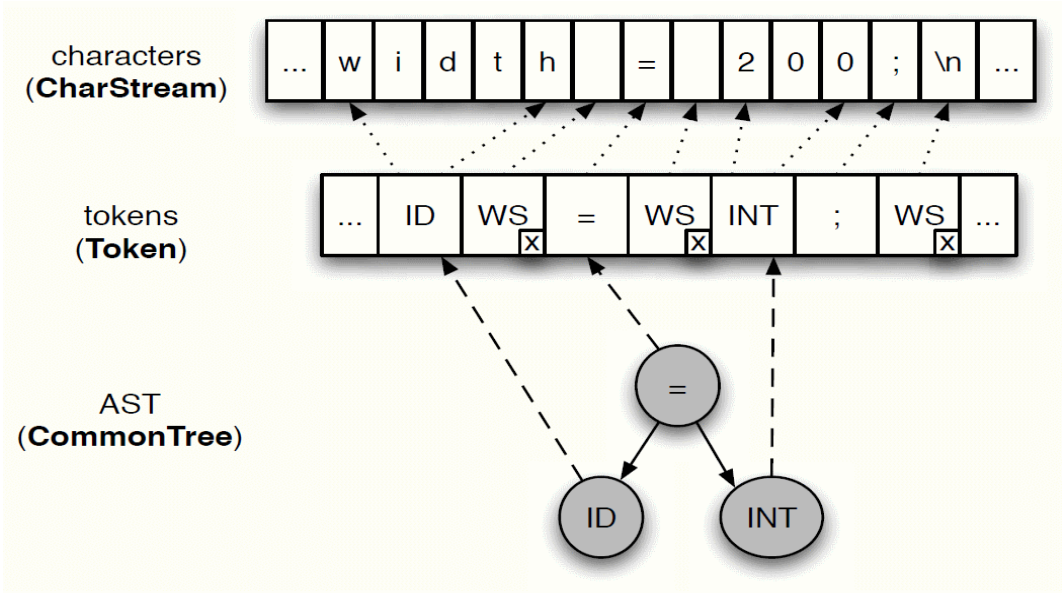

Рис. 8. Процесс преобразования потока символов входной строки в AST-дерево

3.3. Пример грамматики ANTLR 3

Ниже приводится ANTLR-грамматика языка MathLang, который уже рассматривался выше.

```
grammar MathLang;

options {
  language=CSharp3;
  output=AST;
  // backtrack=true;
  // memorize=true;
}


tokens {
  UNKNOWN             ;
  PRINT   = 'print'   ;
  INPUT   = 'input'   ;
  BLOCK               ;
  PROGRAM             ;
}


@lexer::namespace { MathLang }
@parser::namespace { MathLang }


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


INPUT: 'input' ;
PRINT: 'print' ;


NUMBER: ('0'..'9')+ ('.' ('0'..'9')+)?
;
IDENT:  ( 'a'..'z' | 'A'..'Z' | '_' )
        ( 'a'..'z' | 'A'..'Z' | '_' | '0'..'9' )*
;
ADD:    '+'     ;
SUB:    '-'     ;
MUL:    '*'     ;
DIV:    '/'     ;
ASSIGN: '='     ;
```

```
group:
  '('! term ')'!
| NUMBER
| IDENT
;


mult: group ( ( MUL | DIV )^ group )*  ;
add:  mult  ( ( ADD | SUB )^ mult  )*  ;
term: add  ;

expr:
  PRINT^ term
| INPUT^ IDENT
| IDENT ASSIGN^ term
;

program: expr*  ;

result: program EOF -> ^(PROGRAM program);

public execute:
  result
;
```

В ANTLR нет необходимости отдельно описывать грамматику для лексического анализатора, отдельно для синтаксического (но при желании можно описывать и отдельно), одна грамматика служит источником для построения и лексического и синтаксического анализаторов.

Грамматика начинается с задания имени грамматики после ключевого слова **grammar**. Имя грамматики обязательно должно совпадать с именем файла, в котором грамматика описана.

Далее идет секция **options**, в которой описываются общие параметры для всей грамматики, в частности, язык (**language**), на котором должен быть сгенерирован код лексического и синтаксического анализаторов, и тип выходного результата (действий, которые должен проделать парсер) – **output**. Полный список параметров, который можно использовать в секции options, см в документации к ANTLR.

Также полезной может стать опция **backtrack** (= true, значение по умочанию – false), которая меняет алгоритм выделения нужной альтернативы в правилах разбора с LL(*)-автомата, где для однозначного выделения нужной альтернативы поток токенов просматривается вперед на нужное кол-во элементов, на алгоритм с откатами (протестировали нужную альтернативу, не получилось – тестируем вторую и т.д.). Синтаксический анализатор,

построенный с опцией backtrack, менее эффективен. Но описать грамматику ANTLR с данной опцией проще, важно только помнить, что в этом случае порядок перечисления альтернатив становится важным.

В случае включения возвратов (опция backtrack) также можно включить «запоминание» применения правил к разным позициям (опция **memorize**), что теоретически может уменьшить время разбора входной строки, но увеличить размер требуемой для разбора памяти.

В секции **tokens** описываются токены, которым в конкретном синтаксисе разбираемого языка ничего не соответствует, но которые необходимы для построения AST-дерева.

При построении AST-деревьев узлы дерева строятся на основе токенов. Все токены имеют какой-либо тип. Тип – целочисленное свойство Type у токенов и узлов дерева. Полный набор типов токенов для грамматики складывается из:

- токенов, описанных в секции tokens;
- токенов, соответствующих *лексическим* правилам (про *лексические* и *синтаксические* правила ниже);
- токенов, построенных ANTLR для строк, встречающихся в грамматике.

Чтобы пояснить, о каких типах токенов идет речь, ниже приводится фрагмент кода синтаксического анализатора, полученного из описанной выше грамматики.

```
...
namespace  MathLang
{
[System.CodeDom.Compiler.GeneratedCode("ANTLR", "3.3 Nov 30, 2010
12:50:56")]
[System.CLSCompliant(false)]
public partial class MathLangParser : Antlr.Runtime.Parser
{
  internal static readonly string[] tokenNames = new string[] {
        "<invalid>", "<EOR>", "<DOWN>", "<UP>", "UNKNOWN", "PRINT",
"INPUT", "BLOCK", "PROGRAM", "WS", "NUMBER", "IDENT", "ADD", "SUB",
"MUL", "DIV", "ASSIGN", "'('", "')'"
  };
  public const int EOF=-1;
  public const int T__17=17;
  public const int T__18=18;
  public const int UNKNOWN=4;
  public const int PRINT=5;
  public const int INPUT=6;
  public const int BLOCK=7;
  public const int PROGRAM=8;
```

```
  public const int WS=9;
  public const int NUMBER=10;
  public const int IDENT=11;
  public const int ADD=12;
  public const int SUB=13;
  public const int MUL=14;
  public const int DIV=15;
  public const int ASSIGN=16;
...
```

Можно видеть, что токенам, описанным в секции tokens, а также токенам, полученным из «лексических» правила, соответствуют именованные типы токенов. Токены T__17 и T__18 были выделены для строк «скобок» в правиле group. Эти имена можно использовать при анализе построенного AST-дерева для выделения отдельных типов токенов.

Секции **@lexer::namespace** и **@parser::namespace** (для синтаксического анализатора достаточно просто указать **@namespace**) позволяют задать пространство имен, в котором будут созданы классы лексического и синтаксического анализаторов (см. фрагмент сгенерированного кода выше).

В секциях **@lexer::header** и **@parser::header** (для синтаксического анализатора достаточно просто указать **@header**) можно включить код, который будет вставлен в самый верх генерируемых файлов (до начала описания классов). В данных секциях, как правило, подключаются нужные для компиляции лексического и синтаксического анализатора пространства имен (если в грамматике встречается код, для которого эти пространства имен требуется подключить), например:

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

В секциях **@lexer::members** и **@parser::members** (для синтаксического анализатора достаточно просто указать **@members**) можно включить код, который будет вставлен в тело классов лексического и синтаксического анализаторов. В данных секциях можно описывать дополнительные поля, свойства, методы и т.д., используемые во включенном коде (*семантических действиях*) во время разбора, например:

```
@members {
  // number format with decimal separator - '.'
  public static readonly NumberFormatInfo NFI =
      new NumberFormatInfo();
}
```

После описания вышеперечисленных секций в грамматике описываются

правила. Все множество правил в ANTLR-грамматике можно разделить на два класса:

- *лексические* правила используются при построении лексического анализатора, каждое лексическое правило порождает свой тип токена;
- *синтаксические* правила используются при построении синтаксического анализатора, новых типов токенов не порождают.

3.4. Лексические правила

Имена лексических правил должны начинаться с заглавной буквы (как правило, принято использовать все заглавные буквы). Как уже было сказано, каждое лексическое правило порождает свой тип токена.

В правой части каждого лексического правила возможны следующие конструкции:

Конструкция	**Описание**
`'string'`	Сопоставление с указанной строкой в текущей позиции. Данной строке будет сопоставлен отдельный тип токена (во время разбора создан токен) без предопределенного имени.
`'a'..'z'`	Сопоставление с любым из символов из интервала между 'a' и 'z' включительно.
`T`	Вызов лексического правила T (T – token).
`{ C# source code; }`	Семантическое действие – вставка кода на языке C# (или другом языке, если в грамматике указан другой **language**). Данная конструкция будет вызваться в во время разбора.
`E1 | E2 | E3`	Сопоставление с одной из альтернатив, заданных в виде конструкций из левой части данной таблицы (E – expression).
`(E)`	Группировка выражений (E – expression).
`E*`	Сопоставление 0 или более раз (E – expression).
`E+`	Сопоставление 0 или более раз (E – expression).
`E?`	Опциональное выражение, может быть, может не быть (E – expression).

Если есть потребность описать какой-то фрагмент лексического правила, который в дальнейшем будет использоваться при описании других лексических правил, то следует воспользоваться конструкцией **fragment**, которая указывает на то, что данное описание не является полноценным лексическим правилом и не порождает свой тип токена. В приведенной выше грамматике лексические правила NUMBER и IDENT можно реализовать с использованием конструкции fragment следующим образом:

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

NUMBER: DIGIT+ ('.' DIGIT+)?
;
IDENT:  ( WORD | '_' )
        ( WORD | '_' | DIGIT )*
;
```

При описании лексических правил необходимо следить, чтобы более частные правила были описаны ранее более общих. Здесь под более общими правилами понимаются правила, которым соответствуют фрагменты текста, соответствующие и более частным правилам, ярким примером являются ключевые слова языка. Любое ключевое слово соответствует описанию идентификатора. Поэтому, если идентификатор будет описан раньше ключевых слов, то лексический анализатор будет понимать все ключевые слова, как идентификаторы, т.к. подбор нужного лексического правила для фрагмента текста ANTLR осуществляет в порядке описания правил в грамматике. К счастью при обработке грамматике с подобного рода ошибками (описание более общих правил ранее более частных) ANTLR их обнаруживает и порождает соответствующую ошибку (код не генерируется).

Отдельно стоит рассмотреть в приведенной грамматике лексическое правило WS, задачей которого является выделение и удаление пробелов, как не значимых для последующего синтаксического анализа. В фигурных скобках задано *семантическое действие*, т.е. фрагмент кода, который будет выполняться во время разбора. В данном случае с помощью конструкции «$channel=Hidden;» указывается, что токены, получаемые данным правилом, являются незначащими и должны игнорироваться синтаксическим анализатором.

На том же самом принципе (указанием, что получаемые токены следует игнорировать) в грамматике ANTLR могут быть описаны комментарии (в стиле языка C++):

```
SL_COMMENT:
  '//' (options { greedy=false; }: .)* '\r'? '\n' {
    $channel=Hidden;
```

```
  }
;
ML_COMMENT:
  '/*' (options { greedy=false; }: .)* '*/' {
    $channel=Hidden;
  }
;
```

В приведенном выше примере опция **greedy**=false выключает «жадность» алгоритма распознавания, таким образом будет распознана минимальная по размеру подстрока с нужным окончанием (перевод строки или «*/»).

3.5. Синтаксические правила

Имена синтаксических правил должны начинаться с прописной (малой) буквы. В левой части синтаксических правил возможны следующие конструкции:

Конструкция	**Описание**
`'string'`	Сопоставление с указанной строкой в текущей позиции. Данной строке будет сопоставлен отдельный тип токена (во время разбора создан токен) без предопределенного имени.
`T`	Вызов лексического правила T (T – token).
`r`	Вызов синтаксического правила r (r – rule).
`{ C# source code; }`	Семантическое действие – вставка кода на языке C# (или другом языке, если в грамматике указан другой **language**). Данная конструкция будет вызваться в во время разбора.
`E1 | E2 | E3`	Сопоставление с одной из альтернатив, заданных в виде конструкций из левой части данной таблицы (E – expression).
`(E)`	Группировка выражений (E – expression).
`E*`	Сопоставление 0 или более раз (E – expression).
`E+`	Сопоставление 0 или более раз (E – expression).
`E?`	Опциональное выражение, может быть, может не быть (E – expression).

В приведенном примере MathLang синтаксическими правила являются

group, mult, add, term, expr, program, result и exceute.

Стоит заметить, что для синтаксических правил, которые будут вызываться из других классов, необходимо указывать область видимости, как правило, public (начиная с версии ANTLR 3.3).

Правила могут быть рекурсивными, причем рекурсия может быть как прямой (в правой части правила происходит обращение к этому же правилу) или косвенной. Однако левая рекурсия, как прямая, т.е. правила вида

```
rule: rule ... ;
```

, так и косвенная, недопустима.

3.6. Построение AST-деревьев

Как уже говорилось, в ANTLR есть встроенные инструменты для построения AST-деревьев.

В простых случаях можно непосредственно в конструкциях описания синтаксиса указать, какие элементы будут становиться главными (вершиной дерева или поддерева) по отношению к другим элементам, а какие вообще не должны попадать в дерево. Для этого используются конструкции '^' (ставится непосредственно после элемента, который должен стать вершиной по отношению к соседним элементам) и '!' (ставится непосредственно после элемента, который не должен попасть в AST-дерево).

Общие правила формирования деревьев следующие. Если никакие из перечисленных выше конструкций не указаны, то вместо дерева из разбираемых токенов формируется последовательность токенов (однако, чтобы эту последовательность программно представить в виде дерева, в качестве вершины создается искусственный элемент в виде токена с типом 0 и текстом «nil»). Если в процессе обработки какого-то правила встречается элемент, помеченный '^', то этот элемент становится главным по отношению к остальным элементам последовательности и последующие элементы подцепляются уже к этому главному элементу. Если встречается еще один элемент, помеченный '^', то уже он становится главным и его потомками становятся предыдущий главный элемент (сохраняя за собой своих потомков) и последующие элементы.

Для того, чтобы лучше понять данный механизм ниже представлена таблица, демонстрирующая получаемые деревья при различных вариантах синтаксических конструкций.

Предполагается, что в представленных в таблице примерах грамматика начинается с:

```
grammar MathExpr;

options {
  language=CSharp2;
  output=AST;
}

tokens {
  PROGRAM;
}

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

A: 'a' ('0'..'9')* ;
B: 'b' ('0'..'9')* ;
C: 'c' ('0'..'9')* ;
D: 'd' ('0'..'9')* ;
E: 'e' ('0'..'9')* ;
F: 'f' ('0'..'9')* ;
```

Разбор строки в приведенных ниже примерах начинается с правила start.

Грамматика	**Входная строка**	**AST-дерево**	**Комментарий**

`start: A B C D ;`	`a b c d`	`nil`
		`├ a`
		`├ b`
		`├ c`
		`└ d`
			Узел nil – искусственный, только для того, чтобы последоватльность токенов представить в виде дерева.

`start: A^ B C D ;`	`a b c d`	`a`
		`├ b`
		`├ c`
		`└ d`
			Токен A становится главным по отношению к соседним элементам.

`start: A B C D^ ;`	`a b c d`	`d`
		`├ a`
		`├ b`
		`└ c`
			Токен D становится главным по отношению к соседним элементам.

`start: A B (A | C)^ D ;`		`c`
		`├ a`
		`├ b`
		`└ d`
			'^' можно указывать не только к конкретному элементу, но и набору альтернатив.

`start: A B (C)^ D ;`	`a b c d`	`nil`
		`├ a`
		`├ b`
		`├ c`
		`└ d`
			Необъяснимое поведение, учитывая предыдущий пример.

`start: A B^ C D^ ;`	`a b c d`	`d`
		`└ b`
		`  ├ a`
		`  └ c`
			Вначале главным элементом стал токен B, а затем токен D (B своих потомков сохранил за собой).

`part1: B C^ ;`
`start: A part1 D ;`
	`a b c d`	`nil`
		`├ a`
		`├ c`
		`│ └ b`
		`└ d`
			Поддерево, возвращаемое part1 никаких преимуществ в start не имеет.

`part1: A B^ ;`
`start: part1 C D^ ;`
	`a b c d`	`d`
		`├ b`
		`│ └ a`
		`└ c`	

`part1: C D^ ;`
`start: A B part1^ ;`
	`a b c d`	`d`
		`├ c`
		`├ a`
		`└ b`
			В случае указания главным элементом дерева, соседние элементы присоединяются к вершине дерева после уже существующих в дереве элементов.

`part1: A B^ ;`
`start: part1^ (C^ D) ;`
	`a b c d`	`c`
		`├ b`
		`│ └ a`
		`└ d`
			Скобки на формирование деревьев в рамках одного правила никак не влияют.

`part1: A B^ ;`
`part2: C^ D ;`
`start: part1 part2 ;`
	`a b c d`	`nil`
		`├ b`
		`│ └ a`
		`└ c`
		`  └ d`	

`part1: A B^ ;`
`part2: part1 C ;`
`start: part2 D ;`
	`a b c d`	`nil`
		`├ b`
		`│ └ a`
		`├ c`
		`└ d`	

`start: A^ B^ C^ D ;`	`a b c d`	`c`
		`├ b`
		`│ └ a`
		`└ d`
			Главным в дереве становится самый последний встреченный элемент, помеченный '^'.

В случае, если указанных возможностей для формирования правильного

AST-дерева не хватает, можно воспользоваться специальным синтаксисом для формирования AST-дерева из распознанных токенов или поддеревьев (см. примеры в таблице ниже).

Грамматика	**Входная строка**	**AST-дерево**	Комментарий

`start: A B C D`
`  -> C ^(B D) ;`
	`a b c d`	`nil`
		`├ c`
		`└ b`
		`  └ d`
			'^' перед скобкой указывает, что первый элемент в скобке станет главным по отношению к последующим

`start: A B C D`
`  ->  ^(B C ^(D A) ;`
	`a b c d`	`b`
		`├ c`
		`└ d`
		`  └ a`	

`start: A B C D ->  ;`	`a b c d`		Дерево не будет построено

`start: A B C D`
`  -> E F;`
	`a b c d`	`nil`
		`├ E`
		`└ F`
			Токены A, B, C, D будут проигнорированы, вместо них в дерево попадут токены E и F, которым нет соответствия во входной строке (поэтому в дереве они показаны заглавными буквами).

`start: a=A B C D`
`  -> $a $a E[$a] ;`
	`a b c d`	`nil`
		`├ a`
		`├ a`
		`└ a`
			Токен A в дереве будет продублирован 3 раза, причем последний раз тип токена будет изменен на E.

`part1: A | B | C | D ;`
`start: x=part1 part1`
`  y=part1 part1`
`  -> $x $y ;`
	`a b c d`	`nil`
		`├ a`
		`└ c`
			Применяется задание синонимов для вызовов правил

`part1: A | B | C | D ;`
`start: x+=part1 x+=part1`
`  y=part1 part1`
`  -> ^($y $x) ;`
	`a b c d`	`c`
		`└ a`
			Несмотря на то, что в $x содержится 2 элемента в дерева попадает только один (оба элемента попадают в дерево в следующем примере)

`part1: A | B | C | D ;`
`start: x+=part1 x+=part1`
`  y=part1 part1`
`  -> ^($y $x*) ;`
	`a b c d`	`c`
		`├ a`
		`└ b`
			'*' после $x влияет на построение дерева. Кроме того без звездочки, если в $x не было бы вообще элементов, произошла бы ошибка.

3.7. Вызов синтаксического анализатора, сборка проекта

Взаимодействие с построенными лексическим и синтаксическим анализаторами показано в следующем примере:

```
using System;

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


namespace MathLang
{
  public class Program
  {
    public static void Main(string[] args) {
      try {
        // в зависимости от наличия параметров
        // командной строки разбираем
        // либо файл с именем, переданным первым параметром,
        // либо стандартный ввод
        ICharStream input =
          args.Length == 1 ?
            (IcharStream) new ANTLRFileStream(args[0])
          : (IcharStream) new ANTLRReaderStream(Console.In);
        MathLangLexer lexer = new MathLangLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        MathLangParser parser = new MathLangParser(tokens);
        ITree program = (ITree)parser.execute().Tree;
        AstNodePrinter.Print(program);
        Console.WriteLine();
        MathLangIntepreter.Execute(program);
      }
      catch (Exception e) {
        Console.WriteLine("Error: {0}", e);
      }
      Console.ReadLine();
    }
  }
}
```

Классы AstNodePrinter и MathIntepreter такие же, как и в примерах в первой части пособия без использования ANTLR. Единственное отличие, надо вставить в описания данных файлов следующие строки для подключения runtime-библиотеки ANTLR и задания синонимов типов:

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

```
using AstNodeType = MathLang.MathLangParser;
using AstNode = Antlr.Runtime.Tree.ITree;
```

3.8. Пример грамматики для разбора подмножества языка PL/SQL

Ниже приводится довольно сложный пример грамматики для подмножества языка PL/SQL, где демонстрируется большинство возможностей и приемов при работе с ANTLR:

```
grammar SL;

options {
  language = CSharp2;
  output = AST;
}


tokens {
  UNKNOWN                   ;

  AND         =  'and'      ;
  OR          =  'or'       ;

  CALL                      ;
  ELEMENT                   ;
  PARAMS                    ;

  RETURN      = 'return'    ;
  EXIT        = 'exit'      ;
  IF          = 'if'        ;
  IFTHEN                    ;
  THEN        = 'then'      ;
  ELSIF       = 'elsif'     ;
  ELSE        = 'else'      ;
  BEGIN       = 'begin'     ;
  END         = 'end'       ;
  LOOP        = 'loop'      ;
  WHILE       = 'while'     ;
  FOR         = 'for'       ;
  IN          = 'in'        ;
  OUT         = 'out'       ;
  RETURNS     = 'returns'   ;
  IS          = 'is'        ;
  DECLS       = 'declare'   ;

  PRINT       = 'print'     ;
```

```
  EXPR_LIST                ;

  INT_IDENT  = 'int'       ;
  IS         = 'is'        ;

  TYPE_DECL  = 'type'      ;
  CONST_DECL = 'constant'  ;
  VAR_DECL                 ;
  FUNC_DECL  = 'function'  ;
  PROC_DECL  = 'procedure' ;

  TYPE_DECL                ;
  CONST_DECL               ;
  VAR_DECL                 ;
  FUNC_DECL                ;
  PROC_DECL                ;

  BLOCK                    ;
  PROGRAM                  ;
}

@lexer::namespace { SL }
@parser::namespace { SL }


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


INT:    '0'..'9'+                                ;
IDENT:  ( 'a'..'z' | 'A'..'Z' | '_' )
        ( 'a'..'z' | 'A'..'Z' | '_' | '0'..'9' )*  ;

ADD:      '+'   ;
SUB:      '-'   ;
MUL:      '*'   ;
DIV:      '/'   ;

GE:       '>='  ;
LE:       '<='  ;
NEQUALS:  '<>'  ;
EQUALS:   '='   ;
GT:       '>'   ;
LT:       '<'   ;

ASSIGN:   ':='  ;
```

```
group:
  '('! term ')'!
| INT
| IDENT '(' params_ ')'  ->  ^( CALL IDENT params_ )
| IDENT
;


mult:       group ( ( MUL | DIV )^ group )*  ;
add:        mult  ( ( ADD | SUB )^ mult  )*  ;
compare:    add   ( ( GE | LE | NEQUALS | EQUALS | GT | LT )^ add )?
;
and_logic:  compare   ( AND^ compare    )*  ;
or_logic:   and_logic ( OR^  and_logic  )*  ;

term:       or_logic  ;

params_:    ( term ( ',' term )* )?  ->  ^( PARAMS term* )  ;


assignOrCall:
  IDENT (
    '(' params_ ')' (
      ASSIGN term
        ->  ^( ASSIGN ^( ELEMENT["[]"] IDENT params_ ) term )
    |
        ->  ^( CALL IDENT params_ )
    )
  | ASSIGN term  ->  ^( ASSIGN IDENT term )
  |              ->  ^( CALL IDENT PARAMS )
  )
;

ifThenPart: IF term THEN exprList  ->  ^( IFTHEN term exprList )  ;
elsifThenPart: ELSIF term THEN exprList -> ^(IFTHEN term exprList);
elsePart: ELSE^ exprList ;
ifThenElse:
  ifThenPart elsifThenPart* elsePart? END IF
    ->  ^( IF ifThenPart elsifThenPart* elsePart? )
;

expr:
  assignOrCall
| RETURN^ term
| EXIT
| ifThenElse
| LOOP^ exprList END! LOOP!
```

```
| FOR^ IDENT IN! term '..'! term LOOP! exprList END! LOOP!
| WHILE^ term LOOP! exprList END! LOOP!
| block[false]
| PRINT^ term
;

exprList:  ( expr ';'+ )*  ->  ^( EXPR_LIST[".."] expr* )  ;


typeDecl:
  TYPE_DECL IDENT IS INT_IDENT ( '(' add ( ',' add )* ')' )?
  ->  ^( TYPE_DECL IDENT ^( INT_IDENT ^( PARAMS add+ )? ) )
;

type:  INT_IDENT | IDENT  ;

constDecl:
  IDENT CONST_DECL type ASSIGN term
    ->  ^( CONST_DECL["const"] IDENT type term )
;
varDecl:    IDENT type  ->  ^( VAR_DECL["var"] IDENT type )  ;

paramDesc
@init { string paramType = ""; }
:
  IDENT
  ( IN  { paramType += $IN.text;  } )?
  ( OUT { paramType += $OUT.text; } )?
  type
  (
    { paramType.Length > 0 }?  ->  ^( IDENT type
UNKNOWN[paramType] )
  |                            ->  ^( IDENT type )
  )
;
paramsDesc: ( paramDesc ( ',' paramDesc )* )?
               ->  ^( PARAMS paramDesc* )  ;
noParamsDesc:   ->  PARAMS  ;

funcDecl:
  FUNC_DECL IDENT
  (
    '(' p1=paramsDesc ')'
  | p2=noParamsDesc
  )
  RETURNS type
  IS
  decls
  BEGIN exprList END
```

```
    ->  ^( FUNC_DECL IDENT type $p1? $p2? decls  exprList )
;

procDecl:
  PROC_DECL IDENT
  (
    '(' p1=paramsDesc ')'
  | p2=noParamsDesc
  )
  IS
  decls
  BEGIN exprList END
    ->  ^( PROC_DECL IDENT $p1? $p2? decls exprList )
;

decl:
  typeDecl
| constDecl
| varDecl
| funcDecl
| procDecl
;

decls:  ( decl ';'+ )*  ->  ^( DECLS["declare"] decl* )  ;

block [bool program]:
  DECLS decls BEGIN exprList END
  (
    { program }?  ->  ^( PROGRAM decls exprList )
  |               ->  ^( BLOCK   decls exprList )
  )
;

program:  block[true] ';'!+  ;


result:  program  ;

execute:
  result
;
```

Приведенная грамматика вполне успешно справляется с разбором следующего примера:

```
declare
  type NUM is int;
  C constant NUM  :=10*5 -   90;
```

```
  V int;

  function F(A in int)
  returns NUM
  is

    procedure P(A in NUM, B int, C in out int)
    is
    begin
      TTT();
    end;

  begin
  end;

begin
  T (54, C+3, C) := 33;
  B := B + 3 or 8 or 7>8 and (6=8)=9;

  if 5=5 then A;
  elsif 4 =4 then B;
  elsif 3 = 3 then C;;;;
  else D;
  end if;
  while   C>50 and TTT loop
    C := C -2;
  end loop;
  declare0;
  loop
    for i_INDEX in 1..C-10 loop
      null;
    end loop;
  end loop;
  declare
  begin
  end;

  print 4;
end;
```

3.9. Семантические действия

Семантические действия используются, как правило, когда вместо построения AST-дерева, необходимо производить какие-либо вычисления непосредственно во время разбора.

Ниже приводится пример грамматики ANTLR, в которой непосредственно во время разбора происходит вычисление арифметического

выражения:

```
grammar MathExpr;

options {
  language=CSharp3;
  output=AST;
}

@lexer::namespace { MathExpr }
@parser::namespace { MathExpr }

@header {
  using System.Globalization;
}

@members {
  // default number format with "." delimiter
  public static readonly NumberFormatInfo NFI = new
NumberFormatInfo();
}


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


DOUBLE: ('0'..'9')+ ('.' ('0'..'9')+)?
;


number returns [double value]:
  DOUBLE { $value=double.Parse($DOUBLE.text, NFI); }
;

group returns [double value]:
  v1=number { $value=v1.value; }
| '(' v2=add { $value=v2.value; } ')'
;

mult returns [double value]:
  a=group { $value=a.value; } (
    '*' b=group { $value=$value*b.value; }
  | '/' b=group { $value=$value/b.value; }
  )*
;
```

```
add returns [double value]:
  a=mult { $value=a.value; } (
    '+' b=mult { $value=$value+b.value; }
  | '-' b=mult { $value=$value-b.value; }
  )*
;

public execute returns [double value]:
  a=add { $value=a.value; } EOF
;
```

В данном примере в каждом синтаксическом правиле описано возвращаемое значение double value (возвращаемые значения описываются в квадратных скобках после имени правила; правило может иметь как одно, так и несколько возвращаемых значений. Собственно возвращаемые значения вычисляются в семантических действиях, где соответствующие переменные имеют префикс '$'.

3.10. Интерграция ANTLR с Visual Studio

Все проекты Visual Studio содираются с помощью инструмента сборки MSBuild, собственно сам проект (*.csproj) в том числе является корректным скриптом сборки для MSBuild. Поэтому в последовательность сборки можно добавить дополнительные шаги, которые будут выполняться до или после компиляции.

Для того, чтобы во время сборки проекта до компиляции вызывался ANTLR для генерации кода лексического и синтаксического анализаторов достаточно заменить в файле проекта (*.csproj) строку

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

заменить (с учетом имени файла грамматики и т.д.) на:

```
<PropertyGroup>
  <!-- Путь к Java (jdk5-jdk7, jdk8 – не подходит!!!) -->
  <JavaHome>C:\Program_Files\Java\jdk7</JavaHome>
</PropertyGroup>

<ItemGroup>
  <AntlrGrammarFile Include="MathLang.g" />
</ItemGroup>

<PropertyGroup>
  <AntlrLexerFile>
```

```
      @(AntlrGrammarFile->'%(RelativeDir)%(Filename)Lexer.cs')
    </AntlrLexerFile>
    <AntlrParserFile>
      @(AntlrGrammarFile->'%(RelativeDir)%(Filename)Parser.cs')
    </AntlrParserFile>
    <AntlrTokensFile>
      @(AntlrGrammarFile->'%(RelativeDir)%(Filename).tokens')
    </AntlrTokensFile>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="$(AntlrLexerFile)">
      <AutoGen>True</AutoGen>
      <DesignTime>True</DesignTime>
      <DependentUpon>@(AntlrGrammarFile)</DependentUpon>
    </Compile>
    <Compile Include="$(AntlrParserFile)">
      <AutoGen>True</AutoGen>
      <DesignTime>True</DesignTime>
      <DependentUpon>@(AntlrGrammarFile)</DependentUpon>
    </Compile>
    <Compile Include="ParserBaseException.cs" />
  </ItemGroup>
  <ItemGroup>
    <Content Include="$(AntlrTokensFile)">
      <AutoGen>True</AutoGen>
      <DesignTime>True</DesignTime>
      <DependentUpon>@(AntlrGrammarFile)</DependentUpon>
    </Content>
  </ItemGroup>

  <Target Name="GenerateAntlrCode" Inputs="@(AntlrGrammarFile)"
          Outputs="$(AntlrLexerFile);$(AntlrParserFile);
                   $(AntlrTokensFile)">
    <!-- Следующая команда записывается на одной строке !!! -->
    <Exec Command="&quot;$(JavaHome)\bin\java&quot; -classpath
lib\java\antlr-3.5.2-complete.jar org.antlr.Tool -message-format
vs2005 @(AntlrGrammarFile)" />
  </Target>
  <Target Name="CleanAntlrCode">
    <Delete Files="$(AntlrLexerFile);$(AntlrParserFile)" />
  </Target>

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

  <PropertyGroup>
    <BuildDependsOn>
       GenerateAntlrCode;$(BuildDependsOn)
    </BuildDependsOn>
```

```
  </PropertyGroup>
  <PropertyGroup>
    <RebuildDependsOn>
      CleanAntlrCode;$(RebuildDependsOn)
    </RebuildDependsOn>
  </PropertyGroup>
```

Оглавление