УДК 519.682.1

РЕАЛИЗАЦИЯ СИНТАКСИЧЕСКИХ РАСШИРЕНИЙ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ НА ПРИМЕРЕ ДИАЛЕКТА ЯЗЫКА JAVA ДЛЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Д. И. Соломатин

Воронежский государственный университет

Поступила в редакцию 12.11.2010 г.

Аннотация. В статье на примере диалекта языка Java для параллеллельных вычислений рассматривается способ реализации синтаксичеких расширений языков программирования в виде трансляторов кода из расширенного языка в базовый. В качестве инструментария для создания таких трансляторов предлагается использовать генератор синтаксических анализаторов, поддерживающий модульность и расширяемость грамматик.
Ключевые слова: языки программирования, параллельные вычисления, синтаксический разбор, расширение синтаксиса, PEG-грамматики, генератор синтаксических анализаторов.
Abstract. The article describes method of programming language syntax extensions realization in form from extended to base syntax code translators on example of Java dialect for parallel computing. For these translators creating it's suggested to use parser generators with grammar modularity and extensibility support.
Key words: programming languages, parallel computing, syntax parsing, syntax extension, PEG-grammars (Parsing Expression Grammars), parser generator.

ВВЕДЕНИЕ

Разработчики программ часто сталкиваются с ситуацией, когда для простого решения какой-либо специфической задачи в используемом языке программирования не хватает средств или выразительных возможностей. И здесь пожелания разработчиков могут варьировать от добавления в существующий язык удобных им элементов «синтаксического сахара», т.е. конструкций, позволяющих записывать элементы программы более кратко и наглядно, до серьезных нововведений с семантикой, отсутствующей в используемом языке. При этом отдельным пунктом можно выделить языки предметных областей (Domain Specific Language, DSL), для части которых также может быть удобным встраивание в базовый язык.

У исследователей в области языков программирования также возникают подобного рода задачи, заключающиеся в потребности опробовать свои идеи на существующем языке программирования без трудоемкого процесса разработки нового языка и инструментария для него.

Решение подобного рода задач заключается в разработке расширений для существующих языков. Понятно, что в зависимости от сложности необходимого расширения, его реализация может быть выполнена разными способами, начиная от разработки библиотеки в рамках существующего языка до разработки нового компилятора/интерпретатора, поддерживающего новые возможности.

Для широкого класса расширений, которые не могут быть реализованы в рамках синтаксиса и семантики существующего языка, возможна реализация, выполненная в виде транслятора исходных текстов программ из расширенного языка в базовый (*сужающего транслятора*). Такой подход часто применяется на практике.

Разработку сужающего транслятора в рамках рассмотренного подхода нельзя назвать простой задачей, при этом данная задача решается разработчиками и исследователями каждый раз заново с использованием разных подходов и приемов. В данной статье на примере разработки расширения языка Java для параллельных вычислений рассматривается технология реализации таких расширений с использованием генератора синтаксических анализаторов PEG-PG, разработанного автором.

Статья состоит из двух логических частей. В первой части рассматривается расширение

языка Java, названное AsyncJava, и правила преобразования кода с AsyncJava в Java. Во второй части кратко рассматриваются возможности PEG-PG для разработки трансляторов на примере реализации транслятора, выполняющего указанные выше преобразования.

ОБЗОР ASYNCJAVA

Сложность разработки параллельных алгоритмов во многом объясняется неестественностью с точки зрения восприятия человеком языковых средств параллельного программирования. Например, библиотека MPI, общепризнанный стандарт в параллельном программировании с использованием механизма передачи сообщений, разработана прежде всего как эффективная техническая реализация, и о простоте ее применения говорить не приходится.

За прототип расширения AsyncJava взят язык Polyphonic C# (Cω) [1–3]. Те же самые идеи используются в MC# [4,5]. Методы в Polyphonic C# и в AsyncJava могут быть как *синхронными*, так и *асинхронными*, выполнение которых не блокирует вызывающую подпрограмму. Вызов асинхронного метода приводит к созданию нового потока вычислений, в рамках которого выполняется тело метода. Такие методы описываются с помощью ключевого слова **async** и не могут явно возвращать значений. Для синхронизации процесса вычислений и обмена данными между потоками используются так называемые *связки* (chord) нескольких методов (листинг 1):

```
    public class Buffer {
  public int get() & public async put(int v) {
  return v;
  }       (1)
}
```

В данном примере метод `get` (первый описанный) является *основным* методом связки, метод `put` – *связанным* и служит для передачи параметров в основной метод. Связанных методов в связке может быть несколько. Тело связки является по сути телом основного метода.

Семантика данного описания означает, что выполнение метода `get` начнется только после того, как будут получены все параметры из связанных методов, т.е. тогда, когда все связанные методы (в примере метод `put`) будут явно или неявно (через обработчики, построенные на основе этих методов) вызваны (как правило, из асинхронных методов).

На листинге 2 приводится типичная ситуация управления ходом вычислений в параллельных алгоритмах, разработанных на языке типа Polyphonic C#.

```
    public class Service {
  public async service(int param) {
   // эмуляция длительных вычислений
   try { Thread.sleep(10000); } catch(Throwable
ign) { }
   int result=param+1;

   returnServiceResult(result);
  }

  public void waitService()
  & private async returnServiceResult(int
result) {
   System.out.println("service() result is
"+result);
  }

  public void test() {
   service(7);   // вызов вычисления подзадачи
   waitService();// ожидание завершения
вычислений
  }

  public static void main(String[] args) {
   new Service().test();
  }       (2)
}
```

Здесь метод `waitService` является *истинно асинхронным*, т.е. таким, который асинхронно выполняет какие-либо вычислительные действия, в отличие от метода `returnServiceResult`, который, хотя и помечен ключевым словом async, никаких действий, кроме передачи в основной метод связки `waitService` значения параметра `result`, не выполняет.

Стоит заметить, что основной метод связки также может быть асинхронным.

ПРЕОБРАЗОВАНИЯ КОДА ASYNCJAVA В JAVA

На примере листинга 2 рассмотрим, как можно преобразовать представленный код на языке AsyncJava в язык Java с сохранением семантики, описанной в предыдущем разделе.

Код связки методов (`waitService` и `returnServiceResult`) может быть преобразован в код (листинг 3):

```
    public class Service {
  ...
  private ThreadBlockingQueue<Integer>
    _vQueue_01=new ThreadBlockingQueue<In
teger>();

  public void returnServiceResult(int result) {
   _vQueue_01.put(result);
  }
  public void waitService () {
    int result=_vQueue_01.get();
    System.out.println("service() result is
"+result);
  }
  ...      (3)
}
```

Таким образом, правила преобразования связок методов могут быть сформулированы следующим образом:

Для каждого параметра из связанного метода заводится отдельный экземпляр блокирующей очереди (имена таким полям класса можно давать в порядке их объявления в коде). Блокирующей очередь понимается в том смысле, что при попытке чтения значения из пустой очереди, выполнение вызывающей подпрограммы блокируется до момента добавления элемента в очередь из другой параллельной подпрограммы. Добавление элементов в очередь никогда не блокируется (связанные методы асинхронны лишь в этом смысле) – непрочитанные элементы могут накапливаться в очереди. В рассматриваемом примере для параметра `result` связанного метода `returnServiceResult` в преобразованный класс добавлено поле `ThreadBlockingQueue<Integer> _vQueue_01` (примитивные типы в данном случае необходимо заменять их объектными аналогами).

Связанные методы разворачиваются в отдельные методы с теми же самыми именами и параметрами и с телом, в котором переданные параметры просто заносятся в соответствующие этим параметрам блокирующие очереди. В примере метод `waitService`.

Перед телом основного метода связки (телом связки) добавляются локальные переменные, соответствующие параметрам связанных методов с именами этих параметров, чтобы обеспечить видимость в преобразованной программе этих параметров из кода основного метода (тела связки). В добавленные переменные заносятся значения из соответствующих блокирующих очередей. Т.о. если в какой-либо из этих очередей нет значений, т.е. еще не вызван соответствующий связанный метод, то выполнение подпрограммы блокируется. Т.о. тело связки не будет выполнено до тех пор, пока не будут вызваны (повторно вызваны) все связанные методы. В примере строка `int v=_vQueue_01.get()`.

Следует отметить крайне важную особенность реализации алгоритма в процессе чтения/записи результатов вычислений из блокирующих очередей (ThreadBlockingQueue<T>). В текущем потоке выполнения алгоритма из любой такой очереди могут быть прочитаны значения, записанные только порожденными (дочерними для данного потока) потоками, или же значения, записанные в очередь текущим потоком ранее. Это ограничение легко понять, если провести аналогию предложенного алгоритма распараллеливания с рекурсивным алгоритмом (пример с обходом дерева как раз и является параллельно-рекурсивным). Без такого ограничения получилась бы ситуация, когда одни ветви рекурсии могли бы в некоторых ситуациях изменять значения переменных в других ветвях на любом уровне вложенности рекурсии. Описанное ограничение обеспечивается реализацией класса ThreadBlockingQueue<T>.

Истинно асинхронный метод `service` можно преобразовать в следующий код (листинг 4):

```
    public class Service {
 ...
 private void _aFunc_1(int param) {
  // эмуляция сложных длительных вычисле-
ний
  try { Thread.sleep(10000); } catch(Throwable
ign) { }
  int result=param+1;

  returnServiceResult(result);
 }
 public void service(int param) {
  final int _param=param;
  ThreadManager.getThread(
   new Runnable() {
    public void run() {
     _aFunc_1(_param);
    }
   }
  ).start();              (4)
 }
```

Таким образом, правила преобразования истинно асинхронных методов могут быть сформулированы следующим образом:

Код метода полностью выносится в новый метод с точно такими же параметрами, как и в исходном асинхронном методе, в рассматриваемом примере пример – метод `_aFunc_1`. Уровень доступа добавленного метода определяется уровнем доступа асинхронного метода (private – для private, protected – для protected и public).

Тело асинхронного метода заменяется вызовом добавленного метода (`_aFunc_1`) в отдельном потоке вычисления путем запуска экземпляра класса java.lang.Thread (в действительности наследника класса Thread), которому передается созданный анонимный экземпляр класса java.lang.Runnable. В соответствии с правилами языка Java для корректной передачи параметров перед вызовом нити приходится заводить их локальные копии с модификатором final, в примере – переменная `_param`.

Во всех приведенных примерах рассматривались только динамические методы, однако, описанные правила вполне применимы и к статическим (static) методам. В последнем случае добавляемые методы и очереди переменных (`ThreadBlockingQueue<T>`) следует в преобразованном коде также объявлять как статические.

В некотором смысле связанные методы можно рассматривать как генераторы событий возврата данных, а основной метод – обработчиком этих событий (всех сразу). Поэтому задачу динамического назначения обработчиков возврата данных для AsyncJava можно решать, как принято в Java, с помощью создания слушателей (Listener).

Предложенные правила преобразования кода AsyncJava сохраняют интерфейс исходных классов в том смысле, что в результирующем коде присутствуют все методы, объявленные в исходном коде, с теми же самыми параметрами. Семантика выполняемых ими действий также не меняется. Т.о. сохраняется возможность создания скомпилированных библиотек (*.jar) классов параллельных алгоритмов.

На данный момент реализована простейшая подсистема времени выполнения, которая организует выполнение асинхронных методов в отдельных потоках операционной системы (через класс java.lang.Thread). В случае многопроцессорного компьютера такая реализация будет действительно параллельной, собственно распараллеливание задач по вычислительным элементам будет выполнятся операционной системой в виде распределения потоков по различным процессам.

Описанный подход также может быть реализован в рамках кластера из нескольких компьютеров. Более того, для этого не придется менять принципы преобразования кода (препроцессор), достаточно будет реализовать только систему времени выполнения. При этом будут использованы альтернативные реализации классов ThreadManager и ThreadBlocking Queue<T>.

ОБЗОР PEG-PG

Для понимания принципов реализации транслятора с помощью PEG-PG, кратко рассмотрим его возможности. PEG-PG представляет собой генератор синтаксических анализаторов для языка Java по PEG-грамматикам [8].

PEG-грамматики (PEG – Parsing Expression Grammars) были предложены в 2002 г. Брайном Фордом (Bryan Ford) в работе [6]. PEG-грамматики базируются на концепциях ЯНРОВ (язык нисходящего разбора с ограниченными возвратами; англ.: TDPL – top-down parsing language) и ОЯНРОВ (обобщенный ЯНРОВ; англ.: GTDPL – generalized TDPL). В этих алгоритмах нетерминалы трактуются как процедуры, сообщающие о том, обнаружена или нет подходящая цепочка входных символов.

Запись PEG-грамматик во многом похожа РБНФ. Принципиальным отличием от других типов грамматик является использование приоритетного выбора альтернатив (оператор «/», например A / B / C) в отличие от произвольного выбора (оператор «|», например A | B | C), применяемого в других типах грамматик. При разборе входной строки альтернативы тестируются в порядке их записи и всегда применяется первая успешная альтернатива, последующие альтернативы не тестируются. Ниже приводится запись возможного в PEG-грамматике правила с применением конструкции приоритетного выбора (листинг 5 – в данном примере и в примерах ниже подчеркиванием выделены нетерминалы):

```
    IfThenElse:
  “if” “(“ Expr “)” Stat “else” Stat
/ “if” “(“ Expr “)” Stat
;          (5)
```

Как видно из приведенного примера, при описании языка с помощью PEG-грамматик фазы лексического и синтаксического анализа не разделяются.

Второй особенностью PEG-грамматик являются синтаксические предикаты: И-предикат (оператор &, например &A) и НЕ-предикат (оператор «!», например !A). Синтаксические предикаты позволяют тестировать часть строки на соответствие синтаксической конструкции, не «разбирая» эту часть. Ниже приводится запись возможного в PEG-грамматике правила с применением НЕ-предиката (листинг 6):

```
~Identifier: !Keyword !BooleanLiteral
!NullLiteral
      Letter LetterOrDigit*
;    (6)
```

С помощью PEG-грамматик можно задать правила разбора для всех детерминированных контекстно-свободных (КС) языков с концевыми маркерами (т.е. для любых LL и LR языков) и благодаря возможности ограниченных возвратов даже некоторые не КС-языки [7]. С практической точки зрения PEG-грамматики крайне удобны для описания синтаксиса языков программирования, т.к. по своей сути не допускают неоднозначностей в разборе в отличие от других используемых на практике формализмов.

Отсутствие неоднозначностей в PEG-грамматиках позволяет применять к таким грамматикам технику расширения, основанную на переопределении правил (подобно перегрузки виртуальных методов в ООП-языках). Для уже рассмотренного правила `IfThenElse` такое переопределение могло бы выглядеть, например, следующим образом (листинг 7):

```
IfThenElse:
 "if" "(" Expr ")" Stat ( "elsif" "(" Expr ")" Stat
)+ "else" Stat
/ #prev
;        (7)
```

Здесь `#prev` указывает на предыдущую версию объявления правила `IfThenElse`, а конструкция `( ... )+` означает применение выражения в скобках один или более раз.

В PEG-PG поддерживается описанная техника расширения грамматик.

На практике от синтаксического анализатора требуется не только разобрать входную цепочку символов, но и совершить определенные действия по ходу выполнения такого разбора, например, построить синтаксическое дерево программы и т.п. В PEG-PG для этого можно использовать *семантические вставки* кода, которые могут встречаться в любом месте в правой части правил разбора. Каждое правило может возвращать значение какого-либо типа данных, которое может быть прочитано в локальную или глобальную переменную в месте применения правила. Переменные также можно связывать с составным PEG-выражением, в этом случае в переменную помещается строковое значение, продуцируемое данным выражением в применении к входной строке. В общем случае таким значением становится фрагмент входной строки, разобранный данным выражением. В листинге 8 приводится пример, демонстрирующий применение описанных возможностей:

```
    <double> ~number:
 $s=( digit+ ( '.' digit+ )? ) <%
$result=Double.parseDouble($s); %>
;          (8)
```

В данном примере правило number возвращает значение типа double, в правой части строковая переменная `$s` связывается с фрагментом входной строки, соответствующий шаблону `digit+ ( '.' digit+ )?`. Выражение `<% $result=Double.parseDouble ($s); %>` является семантической вставкой кода, в которой вычисляется результат, возвращаемый данным правилом.

Как было сказано, в общем случае в процессе разбора входной строки продуцирует выходная строка, в точности совпадающая с входной. Однако на данный процесс можно повлиять, пропустив какие-либо фрагменты входной строки или же добавив к выходной строке фрагменты, которых нет во входной. Для этого предусмотрены конструкции `#skip( ... )#` и `#out( ... )#`. Фрагмент входной строки, разобранный PEG-выражением, записанным внутри конструкции `#skip( ... )#`, не попадает в выходную строку. Конструкция `#out( ... )#` добавляет к выходной строке произвольный фрагмент текста. Использование данных конструкций удобно для создания source-to-source трансляторов.

РЕАЛИЗАЦИЯ ТРАНСЛЯТОРА ASYNCJAVA В PEG-PG

В данном разделе рассматривается непосредственно реализация транслятора кода с

языка AsyncJava в Java. Данная реализация выполнена в виде расширения PEG-грамматики языка Java 1.5. PEG-грамматика языка Java разработана в соответствии со спецификацией языка [9] и протестированная на корректность на большом количестве исходных кодов открытых Java-проектов.

Для реализации расширения AsyncJava (листинг 9) понадобилось переопределить всего лишь 3 из более чем 100 правил в исходной грамматике языка Java, при этом только одно правило `ClassMemberDecl` расширяет синтаксис языка.

```
    <%!
 // класс, описывающий формальный параметр
 private class FormalParamDescr { ... }

 // класс, описывающий метод
 // (истинно асинхронный метод или связку методов)
 private class MethodDescr { ... }

 // описание предыдущего метода (нужно для правильной индексации)
 private MethodDescr prevMainMethodDescr=null;
%>


// запрет распознавания "async" как идентификатор (тип)
~Identifier: !"async" #prev ;

ClassBodyDeclaration:
 // переменные $$name – глобальные в отличие от локальных $name
 $$modifiers=( Modifier* ) ClassMemberDecl / #prev
;

ClassMemberDecl:
 #prev // асинхронный метод или связку методов #prev не распознает
/
 <%
  // объявляется здесь, иначе за пределами #skip( )# не будет видна
  String code;
 %>
 #skip(
  $tp=( TypeParameters? )
  $type=( "async" / "void" / Type )
  $mnfp=( MethodNameAndFormalParameters ) $a=( ( '[' ']' )* )
  <%
   // формирование описания основного метода
    MethodDescr mainMethodDescr=
     MethodDescr.parseMethodSignature($mnfp);
    mainMethodDescr.setModifiers($$modifiers);
    mainMethodDescr.setTypeParameters($tp);
    $type+=$a;
    mainMethodDescr.setType($type);
  %>
  (
   '&' $modifiers=( Modifier* ) "async"
     $mnfp2=( MethodNameAndFormalParameters )
    <%
     // формирование описания связанного метода
      MethodDescr linkedMethodDescr=
       MethodDescr.parseMethodSignature($mnfp2);
      linkedMethodDescr.setModifiers($modifiers);
      mainMethodDescr.addLinkedMethod(linkedMethodDescr);
    %>
  )*
  $th=( Throws? ) $b=( Block )
  <%
   // формирование описания основного метода
    mainMethodDescr.setThrows($th);
    mainMethodDescr.setMethodBody($b);

    // генерация замещающего кода
    mainMethodDescr.index(prevMainMethodDescr);
    code=mainMethodDescr.generateCode();

    prevMainMethodDescr=mainMethodDescr;
  %>
 )#
 #out( code )# // вывод замещающего кода (за пределами #skip( )#)
;          (9)
```

Как видно из приведенной грамматики, в правило `ClassMemberDecl` добавляется еще одна альтернатива, цель которой – разбор опи-

сания асинхронных методов и связок методов. PEG-выражение, соответствующее такому описанию, полностью размещено внутри конструкции `#skip( ... )#`. Таким образом, объявления асинхронных методов и связок методов напрямую в выходную строку не попадают. Вместо этого в коде семантических вставок (выделены серым фоном) строится преобразованный код на языке Java, который, в свою очередь, с помощью конструкции `#out ( code )#` добавляется к выходной строке.

Для того, чтобы на базе грамматики Java и грамматики расширения AsyncJava получить работающий транслятор, достаточно в грамматике транслятора последовательно подключить соответствующие грамматики (листинг 10):

```
    // подключение базовой грамматики языка
#include “javaspec.peg” ;
// подключение грамматики расширения
#include “asyncjava.peg” ;

<%!
  public String getPreparedSource() {
    return $$preparedSource;
  }
%>

// переопределение правила start для сохранения
// текста разобранной и модифицированной программы
start: $$preparedSource=( #prev ) ;  (10)
```

ЗАКЛЮЧЕНИЕ

Применение генератора синтаксических анализаторов, поддерживающего модульность и расширяемость грамматик, значительно упрощает реализацию расширений языков программирования. При этом появляется возможность комбинировать различные расширения в рамках одного транслятора путем подключения соответствующих грамматик расширений.

СПИСОК ЛИТЕРАТУРЫ

1. *Benton N., Cardelli L., Fournet C.* Modern concurrency abstractions for C#, - draft submitted to ACM Transactions on Programming Languages and Systems, July 2002 (http://research. microsoft.com/en-us/um/people/nick/polyphony/polyphonytoplas-final.pdf)

2. Introduction to Polyphonic C# (http://research.microsoft.com/~nick/polyphony/intro.htm)

3. Cω-Samples, Concurrency Extensions Tutorials (http://research.microsoft.com/en-us/ um/cambridge/projects/comega/doc/comega_tutorials_concurrency_extensions.htm)

4. MC# Home page (http://u-pereslavl.botik.ru/~vadim/MCSharp/)

5. *Сердюк Ю. П.* Введение в параллельное программирование на языке MC# (http://window.edu.ru/ window_catalog/files/r41715/cluster_intro.pdf)

6. *Ford B.* Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking - Master’s Thesis Massachusetts Institute of Technology (http://pdos.csail.mit.edu/~baford/packrat/thesis/ thesis.pdf)

7. *Ахо А.* Теория синтаксического анализа, перевода и компиляции / А. Ахо, Дж. Ульман. – М.:Мир, 1978, т. 1

8. *Львович Я.Е.* Методы и алгоритмы при построении генератора лексико-синтаксических анализаторов для PEG-грамматик / Я.Е. Львович, Д.И. Соломатин // Вестник Воронежского государственного технического университета. – 2008. – т. 4, № 3. – С. 13–17.

9. The Java Language Specification, Third Edition (http://java.sun.com/docs/books/jls/)

Соломатин Дмитрий Иванович – ассистент кафедры Программирования и Информационных Технологий Воронежского Государственного Университета, тел. (4732)208-470

Solomatin Dmitry Ivanovich – Assistant of department of Programming and Information Technologies, Voronezh State University, tel. (4732)208-470