УДК 004.423.25

ОПИСАНИЕ КОНТЕКСТНО-СВОБОДНЫХ ГРАММАТИК В ФОРМАТЕ ДАННЫХ JSON ДЛЯ ГЕНЕРАТОРОВ СИНТАКСИЧЕСКИХ АНАЛИЗАТОРОВ

О. К. Осипов[1]

Московский государственный технический университет им. Н. Э. Баумана

[1] osipovok@student.bmstu.ru

Аннотация

Рассмотрены варианты представления контекстно-свободных грамматик, предлагаемые средствами генерации синтаксических анализаторов. Приведён анализ существующих решений. Предложен новый формат описания грамматики. Дано описание грамматики в виде JSON-документа. Разработана концепция нового генератора, основанная на формате данных JSON для контекстно-свободных грамматик. Описана схема построения анализатора на основе концепции.

Ключевые слова*: JSON-документ, контекстно-свободные грамматики, лексема, форма Бэкуса-Наура, дерево разбора, терминальные символы (токены), конечный детерминированный автомат, парсер, Parglare, ANTLR.*

ВВЕДЕНИЕ

На сегодняшний день существует множество программных средств, которые строят синтаксический анализатор (именуемый также разборщиком или парсером) по данной им контекстно-свободной грамматике. Хотя большинство из них берет за основу дополненную нотацию Бэкуса-Наура [1] (сокр. на англ. ABNF, Augmented Backus Naur Form), некоторые добавляют свои расширения во входной формат грамматики, что ведёт к увеличению сложности и времени для его освоения. Кроме того, не все средства строго следуют каждому предписанию в стандарте ABNF.

В данной статье предложена концепция описания грамматик на основе формата данных JSON (JavaScript Object Notation – нотация объектов JavaScript) [2], [3]. Этот формат определяет структуру и возможное содержимое передава-

емых данных между приложениями в различных информационных системах (начиная от простого веб-сервиса и заканчивая системами управления баз данных и электронных библиотек). Его легко изучить и достаточно просто реализовать в рамках передачи данных между информационными системами. Он широко распространён во множестве современных веб-сервисов. Описано применение концепции к разработке генератора синтаксического разбора грамматик.

ОСОБЕННОСТИ СУЩЕСТВУЮЩИХ РЕШЕНИЙ

Краткое описание дополненной нотации Бэкуса Наура и контекстно-свободной грамматики

Согласно определениям [4, с. 105, 106] и [4, с. 111], контекстно-свободной грамматикой называется четвёрка *G =* (*T, N, P, S),* где

(1) *N* – конечное множество *нетерминальных символов,* или *нетерминалов* (иногда называемые синтаксическими переменными);

(2) *T* – не пересекающееся с N конечное множество *терминальных символов,* или *терминалов* (иногда называемых *словами* или *токенами*);

(3) *P* – конечное подмножество множества

$$N \times (N \cup T)^*$$

(элемент (a, b) множества *P* называется *правилом* (или *продукцией*) и записывается в виде: a -> b);

(4) *S* – выделенный символ из *N,* называемый *начальным* (или *стартовым*) *символом.*

Здесь и далее правило вида $A \to e$ будет означать правило пустой строки, если не указано другое.

Дополненная нотация Бэкуса Наура определяет правила следующим образом: *Rule = elements crlf,* где:

- *Rule* – имя правила (лексического или синтаксического), представленное в виде последовательности символов. Имя правила не чувствительно к регистру, т. е. имена Rule, rulE, RULE определяют одно и то же правило;
- ***=*** – метасимвол, отделяющий имя правила от его тела;

- *elements* – непустая последовательность грамматических символов, т. е. имён правил или *терминальных сущностей*. *Терминальные сущности* представлены в виде последовательности символов, заключенных в двойные кавычки, которые кодируются в неотрицательные целые числа. Следовательно, данную последовательность можно записать в виде последовательности целых положительных чисел, разделённых между собой точкой («.») либо одиночным пробелом. Для указания основания системы счисления каждый код символа предваряется одним из перечисленных ниже префиксов:
  - *%d* – десятичное основание (10);
  - *%b* – двоичное основание (2);
  - *%x* – шестнадцатеричное основание (16);
- crlf – символ конца строки (символ возврата каретки, следующий за переводом строки).

Каждое правило может определять либо продукцию для нетерминала в контекстно-свободной грамматике, либо шаблон (последовательность символов) для терминального символа. Правило для стартового символа *S* определяется в первую очередь. Имя правила, правая часть которого содержит только терминальные сущности, является терминальной сущностью. Множество всех терминальных сущностей составляет множество терминалов грамматики *T.* Множество правил, не являющихся терминальными сущностями, составляет множество продукции грамматики (их имена составляют множество нетерминалов грамматики *N)*. Имя первого определённого правила является стартовым символом грамматики.

Нотация дополнительно определяет несколько операторов:

- “*m***n*R” – оператор повторения, где *m* и *n* – целые положительные числа. Он предшествует имени грамматического символа R (имени правила или терминальной сущности). Означает: “R следует не менее *m* раз и не более *n* раз”. Например: запись «*1*3”+”*» означает – следует “+” или “++” или “+++” (т. е. 1, 2 или 3 раза следует символ “+”). Если *m* не указано, то оно равно 0. Если *n* не указано, то оно равно бесконечности. Таким образом, запись «**1R*» озна-

чает 0 или 1 раз идёт символ R, а запись «1*R» означает 1 раз или более идёт символ R;

- Группирующие круглые скобки “(“, “)” – используются для группировки грамматических символов в сочетании с оператором повторения. Оператор повторения предшествует открывающейся скобке. Для того чтобы использовать скобки в качестве терминальных символов, достаточно заключить их в двойные кавычки;
- Квадратные скобки “[“, “]” – используются для указания выборочной последовательности грамматических символов. Это не что иное, как сокращение выражения “*1(), т. е. вместо записи «*1(foo bar)» можно использовать «[foo bar]»;
- “/” – оператор выбора. Используется между последовательностями элементов одного и того же правила для обозначения альтернативной последовательности (которая следует справа от символа “/”). Может применяться многократно. Например, правило вида abc = “a” / “b” / “c” означает либо “a”, либо “b”, либо “c”.

Анализ входных форматов грамматики существующих генераторов синтаксического разбора

Для анализа формата описания грамматики рассмотрим несколько известных приложений генерации синтаксических анализаторов с открытым исходным кодом:

- Parglare – приложение с открытым исходным кодом, реализующее генератор синтаксических анализаторов типа GLR и LR [5]. Написано на языке программирования Python;
- ANTLR v4 – генератор нисходящих анализаторов (типа LL) для формальных языков [6]. ANTLR преобразует контекстно-свободную грамматику в виде расширенной нотации Бэкуса Наура в программу на C++, Java, C#, JavaScript, Go, Swift, Python. Реализован на языке Java;
- GNU Bison – программа, создающая синтаксические анализаторы по переданному ей описанию грамматики [7]. Реализована на языке C.

Все перечисленные программы берут за основу синтаксиса описания грамматик форму Бэкуса Наура. Однако каждая программа имеет свои собственные интерфейс и функционал для работы с описанием грамматики. Кроме того, каждая программа по-своему задаёт структуру входного файла грамматики, имеет свои расширения для формы Бэкуса Наура. В следующих подразделах перечислены основные элементы синтаксиса описания грамматики для каждой указанной программы. Для сравнения синтаксиса описания грамматики использована простая грамматика арифметических выражений, которая описана ниже в формате ABNF [1]. Терминальные сущности, представленные в виде литералов строк, выделены красным цветом.

```
S = 1*(E newline)
E = E "+" T / E "-" T / T
T = T "*" F / T "/" F / F
F = "(" E ")" / number
number = 1*digit [ "." 1*digit ]
digit = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"
newline = %x2424
```

Листинг 1. Грамматика арифметических выражении в нотации ABNF.

Синтаксис описания грамматики, предлагаемый программой Parglare

Входной файл описания грамматики имеет расширение *.pg* и состоит из следующих секций, расположенных в указанном ниже порядке:

- Секция инструкций импорта. Она не является обязательной;
- Секция правил, содержащая одно или множество правил грамматики;
- Секция терминалов грамматики “terminals”. Имя секции (terminals) указывается обязательно.

Инструкции импорта принимают имя файла грамматики и позволяют сослаться на его содержимое по имени импортируемой грамматики. Именем грамматики считается по умолчанию имя файла, в котором она написана. Таким образом инструкции импорта позволяют определять одну грамматику в не-

скольких файлах по частям. Синтаксис инструкции выглядит следующим образом:

- import '…/…/myGrammarPart1.pg';

Правило грамматики имеет следующий вид:

- *ruleName: expression ;*

Здесь *ruleName* означает имя нетерминального символа грамматики (синтаксической переменной), а *expression* – последовательность имён нетерминальных или терминальных символов грамматики, разделённых пробелом. Каждое правило заканчивается точкой с запятой (";"). Терминальным символов может быть либо заранее предопределённое имя терминала, либо строка символов, заключенная в двойные кавычки.

Имена терминалов определяются в секции terminals, следующей за всеми правилами грамматики. Определение терминала выглядит следующим образом:

- termName: stringOrRegex ;

Здесь termName – имя терминала, а stringOrRegex – либо строка, заключенная в двойные кавычки, либо литерал регулярного выражения на языке Python (шаблон токена), либо пустая строка. Аналогично правилам грамматики каждое определение терминала заканчивается точкой с запятой. Имя EMPTY зарезервировано и используется как правило, обозначающее пустую строку. Parlage позволяет добавлять собственные лексические распознаватели к терминалам, у которых в правой части (после двоеточия) ничего нет кроме пробелов и одиночного символа двоеточия.

Согласно дополненной нотации Бэкуса Наура [1] каждое правило (синтаксическое и лексическое) должно заканчиваться символом конца строки. В Parlage вместо него используется точка с запятой. Parlage определяет собственный набор операторов, отличный от нотации ABNF. Parlage позволяет повторять определённый грамматический символ (терминал и нетерминал) один или более раз с помощью оператора "+", который должен следовать сразу за именем грамматической сущности, в отличие от нотации, где оператор повторения

определяется между двумя числами, задающими диапазон повторений, перед именем грамматического символа. Символ “?” используется аналогично, но означает повторение 0 или 1 раз. Его можно использовать вместо EMPTY для выборочной последовательности грамматических символов. В качестве оператора выбора выступает символ вертикальной черты “|”. В файле грамматики можно оставлять однострочные комментарии в стиле языка C. Ниже дан пример грамматики арифметических выражений в форме синтаксиса Parglare.

```
S:  E  newline  SE;
SE: E  newline  SE  |  EMPTY;
E:  E  “+”  T |  E  “-“  T  |  T;
T:  T  “*”  F |  T  “/”  F  |  F;
F:  “(“  E  “)”  |  number;

terminals
number:  /\d+(\.\d+)?/;
newline:  /\n/;
```

Листинг 2. Грамматика арифметических выражении в нотации Parglare.

Большим преимуществом данной программы являются модульность грамматик, а также наличие отдельных секций для лексических и синтаксических правил, определяющих продукции и терминалы. Однако синтаксис регулярных выражений был взят из языка программирования Python, и их обработка выполняется на нём, т. е. регулярные выражения привязаны к реализации программы генератора, что является существенным недостатком данного приложения. Вторым недостатком является отсутствие поддержки группировки, определённой в нотации ABNF. Приложение не умеет группировать цепочку грамматических символов. Пара круглых скобок “(” и “)” используются лишь для группировки регулярных выражений, а не грамматических единиц.

Синтаксис описания грамматики, предлагаемый программой ANTLR 4

Файл грамматики имеет расширение .g4 и состоит из следующих разделов:

- Заголовок имени грамматики. Состоит из ключевого слова *grammar* и следующим за ним идентификатора. Заканчивается символом точка с запятой, следующим за идентификатором. Имя грамматики и имя файла должны совпадать;
- Раздел опций (options {…}). Данный раздел не является обязательным;
- Раздел токенов (tokens {…}). В данном разделе перечисляются имена токенов через запятую. Токены являются терминальными символами грамматики, для которых не определены лексические правила. Имена токенов прописные. Данный раздел не является обязательным;
- Раздел инструкций импорта. Не является обязательным;
- Раздел синтаксически управляемых действий (actions {…}). Содержит исполняемый Java-код. Не является обязательным;
- Раздел правил (синтаксических и лексических). Имя для лексического правила набирается в прописном регистре. Имя синтаксического правила должно начинаться со строчной буквы.

Правила грамматики могут быть представлены в следующих видах:

LEXRULE : lexems ;

syntaxRule : elements ;

Здесь *LEXRULE* – имя лексического правила, задающего шаблон для терминала (с именем *LEXRULE*), *lexems* – элементы лексического правила, разделённые пробелами между собой. Элементами лексического правила могут быть последовательности символов, заключённых в одинарных кавычках, имена других лексических правил, или *фрагменты*. Фрагмент – это часть грамматики, которая может быть только частью лексического правила. Фрагменты нельзя использовать в синтаксических правилах. Лексическое правило может стать фрагментом. Для этого необходимо предварить имя правила ключевым словом fragment. Ключевое слово указывается перед именем правила отдельно.

Имя *syntaxRule* обозначает продукцию в грамматике с одноимённым нетерминалом в качестве заголовка и *elements* в качестве тела. Здесь *elements* обозначает последовательность имён правил (любых, как лексических так и синтаксических) и строк символов, заключенных в одинарные кавычки.

Каждое правило заканчивается символом «точка с запятой» и отделяется от другого правила символом новой строки. Первое определённое правило является продукцией для стартового символа грамматики. ANTLR также позволяет группировать грамматические элементы в цепочки, как в нотации ABNF. ANTLR предоставляет собственные операторы повторения, перечисленные ниже:

(1) “+” – оператор повторения 1 или более раз;
(2) “*” – оператор повторения 0 или более раз;
(3) “?” – оператор повторения 0 или 1 раз. (аналогично Parglare).

Каждый из операторов, перечисленных выше, повторяет цепочку грамматических символов, расположенную слева от него. Оператор выбора реализован так же, как в программе Parglare. Ниже описана грамматика арифметических выражений в форме синтаксиса ANTLR.

```
grammar  Expressions;

prog:    (expr NEWLINE)+ ;

expr:  expr ( '*' | '/' ) term   |  term ;

term:    term ('+' | '-' ) factor   |  factor ;

factor:  NUMBER   |  '(' expr ')'   ;

NEWLINE  : [\n];

NUMBER  : [0-9]+(\.[0-9]+)? ;
```

Листинг 3. Грамматика арифметических выражении в нотации ANTLR.

Формат описания грамматики, предлагаемый ANTLR, обладает большим недостатком. В первую очередь, от регистра имени зависит тип правила (определение терминала, или синтаксической продукции) для грамматики. Зависимость от регистра может внести путаницу при создании грамматики. Кроме того, нотация ABNF явно указывает, что имена правил не зависят от регистра букв.

Лучшим решением является определение отдельной секции для лексических правил, как это было сделано в приложении Parglare. Также программа ANTLR не умеет разрешать неявную левую рекурсию, следовательно, она перекладывает устранение левой рекурсии из грамматики на плечи рядового пользователя, в то время как существует алгоритм устранения левой рекурсии из [5, с. 180], позволяющий решить данную проблему.

Достоинством ANTLR является поддержка круглых скобок, группирующих цепочки грамматических символов. Благодаря им можно сократить дублирование, возникающее при перечислении альтернатив синтаксических правил, у которых есть общие части цепочек.

Синтаксис описания грамматики, предлагаемый программой GNU Bison

Входной файл грамматики для приложения GNU Bison имеет имя, оканчивающееся на *.y,* оно состоит из четырёх секций, расположенных в порядке, указанном ниже:

- Секция пролога. Содержит макроопределения, макросы, объявления переменных и функций на языке C. Может отсутствовать;
- Секция определений. В данной секции определяются имена терминальных и нетерминальных символов грамматики. Для объявления терминала необходимо объявить токен с его именем. Имя токена должно быть полностью в верхнем регистре. Токены объявляются инструкцией *%token*. Секция не является обязательной;
- Секция правил. Содержит непустую последовательность определений правил трансляций. Данная секция начинается с заголовка, состоящего из двух символов: ‘%%’. Далее, после заголовка, с новой строки идут определения продукций грамматики. Синтаксис определения аналогичен синтаксису приложения Parglare, но с ограничением на имена нетерминалов – они не должны состоять из одних прописных букв, т. к. все имена токенов должны быть полностью в верхнем регистре. Пустое тело правила (обозначающее пустую строку) обозначается как %empty;

- Секция эпилога. Данная секция следует после секции правил. Для её отделения от секции правил используется тот же заголовок ‘%%’. Не является обязательной;

Ниже описана грамматика арифметических выражений в виде файла грамматики GNU Bison:

```
%token NUMBER
%%
s:  expr  line  se;
se: expr line se  |  %empty;
line:   expr  ‘\n’;
expr:  expr  ‘+’  term  |  expr  ‘-‘  term  | term ;
term: term  ‘*’  factor  |  term ‘/’  factor  | factor ;
factor: ‘(‘  expr  ‘)’  |  NUMBER ;
%%
yylex(){
return  getNumber();
}
```

Листинг 4. Грамматика арифметических выражении в нотации GNU Bison.

Одним из главных недостатков данного формата является отсутствие задания лексических правил для терминальных символов грамматики. Всю работу по сборке последовательности символов в терминалы (в слова) приходится выполнять вручную, с использованием предопределённой функции *yylex*. Она должна вернуть положительное целое число, связанное с определённым токеном. Программа автоматически назначает определённым токенам их коды, так что для возвращения кода достаточно набрать имя токена. Функция также должна сохранить распознанную лексему в переменной *yylval*. В листинге 4 работа функции переложена на функцию *getNumber*, которая собирает число (как целое, так и дробное) в переменную *yylval* и возвращает код токена NUMBER. Ещё одним

существенным минусом является отсутствие поддержки операторов группировки и повторения, определяемых ABNF.

Также программа использует имена токенов (терминалов) в верхнем регистре, что опять может привести к ошибкам в определении имён правил грамматики.

Однако программа способна справляться с леворекурсивными грамматиками, в отличие от приложения ANTLR. Она также способна обрабатывать правую рекурсию. Программа способна работать с генератором лексических анализаторов Lex, позволяя ей определять лексические правила для токенов, задаваемых текущим форматом грамматики.

ФОРМАТ ДАННЫХ JSON ДЛЯ КОНТЕКСТНО-СВОБОДНЫХ ГРАММАТИК

Во всех перечисленных форматах есть определённые особенности, присущие каждому. Однако они не определяют все компоненты контекстно-свободной грамматики в явном виде. В каждом из них используются свои собственные правила, диктующие, какие элементы относятся к определённому компоненту грамматики. Ниже представлен новый формат описания грамматик, лишённый данного недостатка. В нём для каждого компонента определена соответствующая секция.

Контекстно-свободная грамматика состоит из четырёх элементов: множества терминалов *T*, нетерминалов *N*, продукций *P* и стартового символа *S.* Определим для неё JSON-документ, который будет хранить информацию об этих множествах. JSON-объект будет состоять из четырех свойств:

- Свойство “terms” представляет множество терминалов грамматики в виде JSON-объекта, содержащего коллекцию пар *ключ–значение. Ключом* является имя терминала, которое должно быть уникальным в пределах объекта. *Значением* является строка, содержащее регулярное выражение. Синтаксис регулярного выражения был взят из [4, с. 124] и дополнен в следующем параграфе настоящей статьи;
- Свойство “nonTerms” представляет JSON-массив, состоящий из строк, обозначающих имена нетерминалов грамматики;

- Свойство “start” представляет собой строку, обозначающую имя нетерминала из массива “nonTerms”, который является стартовым символом грамматики;
- Свойство “productions” хранит продукции грамматики в JSON-массиве. Элементами массива являются JSON-объекты с единственным свойством. Имя такого свойства должно присутствовать в массиве nonTerms (т. е. быть нетерминалом грамматики). Значением такого свойства являются либо массив грамматических символов, либо литерал null, либо одиночный грамматический символ, выраженный строкой. Все грамматические символы, составляющие тела продукций, должны быть выражены в качестве строк, обозначающих имена терминалов или не терминалов (т. е. либо являются элементами массива nonTerms, либо свойствами объекта terms). В случае, когда тело продукции состоит из более чем одного грамматического символа, необходимо использовать JSON-массив. В массиве перечисляются строки (имена терминалов или нетерминалов) в определённом порядке. Литерал null можно использовать для обозначения тела продукции, порождающей пустые строки, вместо имени терминала, значение которого в объекте *terms* равно null. Для создания альтернатив к определённому правилу с именем N необходимо создать новый JSON-объект в данном массиве со свойством, имеющим такое же имя N.

Во многих языках программирования зарезервированы специальные ключевые слова. Можно определить дополнительное, необязательное свойство в объекте грамматики с именем *keywords*, значением которого будет массив строк, представляющих ключевые слова.

Дополнения к регулярным выражениям

Регулярные выражения снабжены дополнительными операциями:

- “+” – положительное замыкание Клини. Означает повторение шаблона 1 или более раз;
- [a-z] – классы символов. Являются сокращением многократного использования оператора объединения “|”. Например, указанная запись «[a-z]» означает «(a|b|c|…|z)»;

- “_” – зарезервированное регулярное выражение, которое обозначает любую строку из одного символа, т. е. любой одиночный символ;
- “@” – экранирующий символ. Любой одиночный символ, который следует за ним, воспринимается буквально (как операнд). Чтобы написать выражение для строки из символа “@”, достаточно его продублировать – “@@”. Во многих языках программирования в синтаксисе регулярных выражений для аналогичных действий используют символ обратного слеша ”\”. Но поскольку стандартом [3] определено его использование при разборе JSON-документов, в целях совместимости со стандартом для регулярных выражений был определён экранирующий символ собаки “@”;
- JSON-литерал null обозначает регулярное выражение для множества пустых строк.

При использовании последовательности числовых кодов символов налагается ещё одно ограничение: числовые коды со значениями 0 и 1 зарезервированы для символа “_” и символа пустой строки соответственно. При их указании в последовательности нужно понимать, что в данном случае символы будут интерпретироваться как “_” (в случае 0) или как маркер пустой строки (в случае 1).

Таким образом, опираясь на определённый JSON формат грамматики из предыдущего параграфа и синтаксиса регулярных выражений с вышеперечисленными дополнениями и ограничениями грамматика арифметических выражений представлена следующим образом:

```
{
“terms” : {
        “newline” : "\n",
        “empty” :  null,
        “number” : "[0-9]+.[0-9]+(((E|e)(-|empty)[0-9]+)|empty)",
        “+” : "+",
        “-” : "-",
        “*” : "*",
        “/” : "/",
        “(” : "(",
        “)” : ")"
},
“nonTerms” : ["S", "SE", "E", "T", "F"],
“productions” : [
        {“S“ : ["E", "newline", "SE"] },
        {“SE“ : ["E", "newline", "SE"] }, {“SE“ : "empty" },
        {“E“ : ["E", "+", "T"] }, {“E“ : ["E", "-", "T"]}, {“E“ : "T"},
        {“T“ : ["T", "*", "F"] }, {“T“ : ["T", "/", "F"]}, {“T“ : "F"},
        {“F“ : ["(", "E", ")"] }, {“F“ : "number"}
        ],
        “start” : "S"
}
```

Листинг 5. Грамматика арифметических выражении в нотации JSON.

ГЕНЕРАЦИЯ ПРОГРАММЫ СИНТАКСИЧЕСКОГО РАЗБОРА ИЗ JSON-ДОКУМЕНТА КОНТЕКСТНО-СВОБОДНОЙ ГРАММАТИКИ

Опишем концепцию генератора синтаксических анализаторов, который будет использовать предложенный JSON-формат.

Описание предметной области

Для составления схемы построения и работы генератора необходимо выделить сущности и их обязанности. Для этого была создана карта сущностей (рис. 6) и на её основе составлен список обязанностей и основных функции, требуемых при создании приложений генераторов разборщиков.

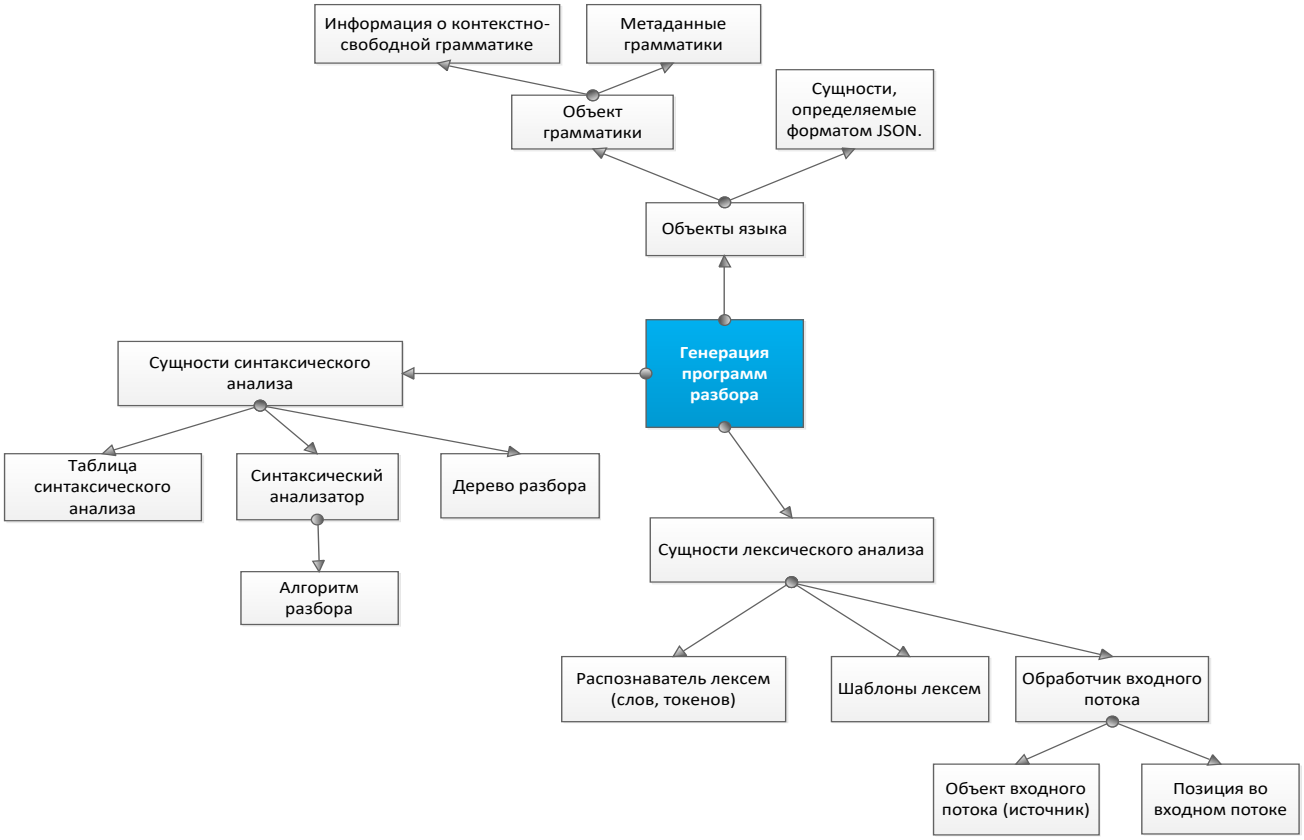

Рисунок 6. Карта основных сущностей генератора программ разбора.

Перечислим основные обязанности приложения:

- Обработка JSON-документов и их представление в виде дерева;
- Создание объекта грамматики:

  - Обработка дерева JSON-документа, хранящего информацию о грамматике;
  - Проверка JSON-документа на соответствие формату, описанному в данной статье;
  - Выделение дополнительной информации, не относящейся к определению контекстно-свободных грамматик, и её сохранение в виде отдельного объекта;
- Преобразование грамматики к виду, пригодному для применения одного из алгоритмов разбора:
  - Устранение левой рекурсии;
  - Устранение пустых правил. Если язык, задаваемый грамматикой, порождает пустые строки, то в грамматике должно присутствовать правило вида $S \to e$, где нетерминал S не встречается в правых частях любых других правил грамматики;
  - Устранение циклов;
- Создание лексического анализатора по шаблонам терминалов грамматики:
  - Чтение из источника (входного потока);
  - Идентификация распознанной лексемы;
  - Оповещение о лексической ошибке в случае невозможности распознать лексему;
- Генерация таблицы синтаксического анализа по одному из алгоритмов, задаваемых пользователями (LL, LR, LALR, SRL, GLR);
- Создание объекта разборщика на основе таблицы синтаксического анализа и лексического анализатора;
- Анализ текстовых файлов построенным синтаксическим анализатором:
  - Выявление синтаксических и лексических ошибок;
  - Построение дерева разбора, в случае отсутствия ошибок;
  - Восстановление при обнаружении ошибок.

Структура генератора программ разбора

Основываясь на карте сущностей и списке обязанностей, представленных в предыдущем параграфе, можно построить набор компонентов и модулей генератора.

Основные структурные модули приложения:

- JsonParser – основной класс, объединяющий процедуры разбора JSON документов. Имеет два метода: *parseFile* и *parseStream*, которые принимают в качестве параметра либо имя файла, либо поток целочисленных, положительных символьных кодов;
- JsonElement – корень иерархии типов, задаваемых стандартом JSON [3]. С помощью него извлекаются значения данных различных типов;
- JsonObject – тип объекта, представляющий структуру и основное содержимое прочитанного JSON-документа. Предоставляет два метода: *getProperty* и *getElement*. Оба метода принимают строку, обозначающую имя свойства. Метод *getProperty* ищет указанное свойство в самом JsonObject без учёта вложенности. Метод *getElement* учитывает вложенные объекты. Оба метода возвращают экземпляр класса JsonElement, который хранит значение данного свойства;
- ILexer – интерфейс взаимодействия с лексическим анализатором, позволяющий получать токены по одному. Определяет единственный метод *recognize*, который принимает в качестве параметра указатель входного потока. ILexer читает лексемы посимвольно с помощью ILexerIO, предоставляя ему всю работу с входным потоком;
- ILexerIO – интерфейс взаимодействия с входным потоком, откуда лексический анализатор читает символы и собирает их в лексемы. В нём определены следующие методы:
  - *ungetch(char c)*: сохраняет символ *c* в буфер;
  - getch() – вернёт символ с буфера, если он не пуст, иначе вернёт текущий символ с входного потока. После этого переместит указатель входного потока на один символ вперёд;

  - getInput() – возвращает текущий символ входного потока. После этого переместит указатель входного потока на один символ вперёд;
- Token – тип объекта, содержащего информацию о распознанной лексеме, выраженную в трёх полях:
  - name – имя символа (распознанной лексемы);
  - value – сама лексема (прочитанная строка символов);
  - type – тип символа. Определяет, является ли символ терминалом, нетерминалом, символом конца потока или ошибки;
- DFALexer – реализация интерфейса ILexer. Основана на архитектуре лексического анализатора программы Lex и детерминированного конечного автомата, который строится по алгоритмам [8, с. 206] и [8, с. 213]. Реализует модуль лексического анализа;
- App – основной класс приложения, инкапсулирующий процесс построения анализатора и синтаксического разбора передаваемых файлов;
- Grammar – тип объекта, представляющий информацию о контекстно-свободных грамматиках. Может включать в себя объект типа GrammarInfo;
- GrammarInfo – тип объекта, содержащий дополнительные данные о грамматике (ключевые слова, терминалы, обозначающие комментарии, т. е. слова, которые надо игнорировать синтаксическому анализатору);
- Parser – абстрактный класс, представляющий синтаксический анализатор. Хранит два объекта: экземпляр класса Grammar и DFALexer. Его подклассы должны реализовать один из алгоритмов синтаксического анализа;
- ParsedTree – дерево разбора, создаваемое объектом типа Parser. Содержит информацию о синтаксической структуре разобранного текста.

Схема работы генератора

Главным классом приложения является *App*. Он отвечает за работу остальных модулей программы. Принимает на вход от пользователя имя файла, содержащее описание грамматики в формате JSON. Далее, при успешном построении анализатора ожидает на вход текстовые файлы для синтаксического разбора. Схема работа данного класса представлена на диаграмме последовательности UML (рис. 7).

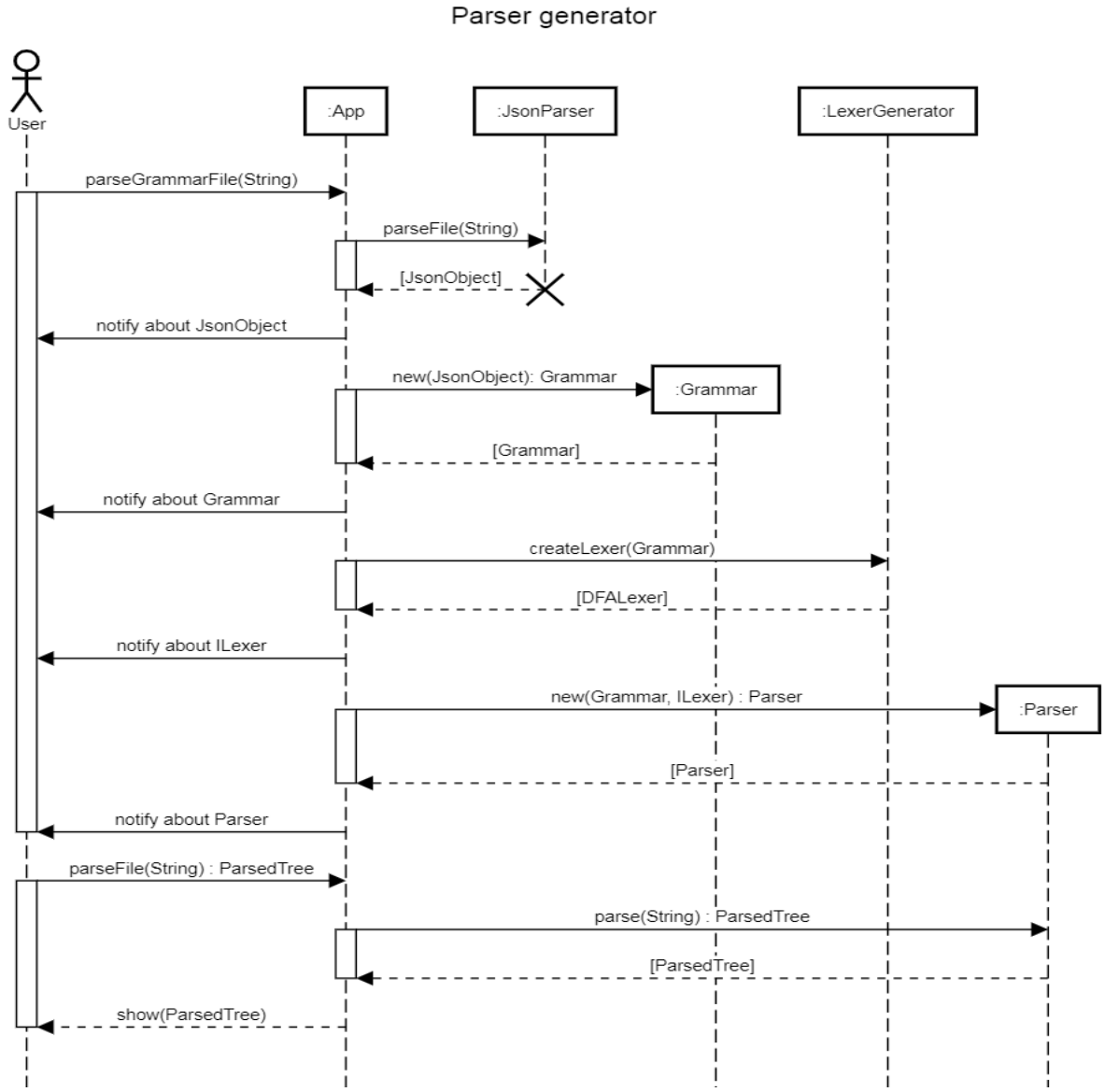

Рисунок 7. Диаграмма последовательности UML для программы генератора синтаксических анализаторов.

ЗАКЛЮЧЕНИЕ

Предложенный формат описания грамматик позволяет упростить процесс создания программы разбора. С помощью формата данных JSON и отдельных секций для каждого компонента грамматики очень легко и просто отделить лексические правила и единицы от синтаксических. Использование данного формата с предложенной архитектурой генераторов синтаксического разбора позволяет сконцентрироваться на разработке контекстно-свободного языка, а разделение обязанностей по чтению и распознаванию лексем в два отдельных подмодуля предоставляет возможность разработчику выбирать тип источника, который можно преобразовать в последовательность символов. Также данный формат позволяет легко расширять или модифицировать язык, определяемый грамматикой.

СПИСОК ЛИТЕРАТУРЫ

1. Standard RFC 5234: Augmented BNF for Syntax Specifications: ABNF. URL: https://tools.ietf.org/html/rfc5234#section-2.1

2. Standard RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format. URL: https://tools.ietf.org/html/rfc8259

3. Standard ECMA 404: The JSON Data Interchange Standard. URL: https://www.json.org/json-en.html, https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

4. *Ахо А., Ульман Дж.* Теория синтаксического анализа, перевода и компиляции: В 2-х томах. Т. 1: Синтаксический анализ. М.: Мир, 1978.

5. Parlage // Github. URL: https://github.com/igordejanovic/parglare

6. ANTLR: ANother Tool for Language Recognition. URL: https://www.antlr.org/

7. GNU Bison: Manual. URL: https://www.gnu.org/software/bison/manual/bison.html

8. *Ахо А.В., Лам М.С., Сети Р., Ульман Дж. Д.* Компиляторы: принципы, технологии и инструментарий, 2-е изд.: Пер. с англ. М.: Вильямс, 2008. 1184 с.

DESCRIPTION OF CONTEXT FREE GRAMMARS IN JSON FORMAT FOR PARSER GENERATORS

Oleg Osipov [1]

Bauman Moscow State Technical University

[1]osipovok@student.bmstu.ru

Abstract

Analysis of various presentations for context free grammars provided with parser generators. A new description format of context free grammars is proposed. Given a representation of context free grammar in JSON format. The concept of a new parser generator based on JSON data format of describing context free grammars is presented. Described a parser generation scheme based on that concept.

Keywords: *JSON-document, context free grammars, lexeme, Backus Naur Form, parsing tree, terminal symbols (tokens), deterministic finite state automata, parser, Parglare, ANTLR.*

REFERENCES

1. Standard RFC 5234: Augmented BNF for Syntax Specifications: ABNF. URL: https://tools.ietf.org/html/rfc5234#section-2.1

2. Standard RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format. URL: https://tools.ietf.org/html/rfc8259

3. Standard ECMA 404: The JSON Data Interchange Standard. URL: https://www.json.org/json-en.html, https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

4. *Alfred V. Aho, Jeffrey D. Ullman.* The theory of parsing, translation and compiling: In 2 volumes. V.1: Parsing. Moscow, Mir Publ., 1978.

5. Parlage // Github. URL: https://github.com/igordejanovic/parglare

6. ANTLR: ANother Tool for Language Recognition. URL: https://www.antlr.org/

7. GNU Bison: Manual. URL: https://www.gnu.org/software/bison/manual/bison.html

8. *Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman.* Compilers: Principles, Techniques and Tools, 2nd ed. Moscow, Addison-Wesley Publ., 2008. 1184 p. (In Russian).

СВЕДЕНИЯ ОБ АВТОРЕ

ОСИПОВ Олег Константинович – студент магистратуры факультета Информатики и систем управления Московского государственного технического университета им. Н.Э. Баумана, направление подготовки – «Программная инженерия».

Oleg Konstantinovich OSIPOV, student of 1 grade of master program of Computer Science and Control Systems department of Bauman Moscow State Technical University.

Email: osipovok@student.bmstu.ru

Материал поступил в редакцию 15 августа 2020 года