Обзор современных средств автоматизации создания синтаксических анализаторов

И. С. Чемоданов
ilia239@tercom.ru

Н. П. Дубчук
dubchuk@tercom.ru

Специализированные средства значительно упрощают процесс разработки синтаксических анализаторов. В настоящий момент существует большое количество таких инструментов, различающихся алгоритмами разбора, средствами восстановления после ошибок и переиспользования описаний грамматик, целевыми языками и т. д. Данная статья содержит обзор наиболее известных средств автоматизации создания синтаксических анализаторов, описание набора критериев для их сравнения и оценку рассмотренных средств относительно этого набора.

Введение

Вопросам построения синтаксических анализаторов по описанию грамматики разбираемого языка посвящена обширная литература [2, 3, 6]. Многие учебники упоминают о таких инструментах и чаще всего приводят в качестве примера YACC [9].

За 30 лет эволюции (если считать от первых вариантов YACC) средства описания трансляций претерпели значительные изменения. Большое влияние на них оказало развитие языков программирования. Любой программист, найдя десятки генераторов анализаторов в сети Интернет, заинтересуется, чем же все они отличаются друг от друга?

В данной статье сначала формулируются рекомендуемые критерии сравнения генераторов, а затем проводится анализ возможностей современных инструментов в соответствии с заданными критериями.

Авторы выражают признательность Я. А. Кириленко за множество полезных советов и критических замечаний, позволивших существенно улучшить содержание статьи.

1. Рассматриваемые инструменты

В настоящее время классические инструменты YACC и LEX[1], а также производные от них[2] вытесняются новыми, более удобными в использовании разработками. Из всего многообразия реализаций классического YACC в данный обзор включен лишь GNU Bison как наиболее совершенный. Но кроме популярных инструментов мы рассматриваем ряд генераторов, заслуживающих, по нашему мнению, особого внимания.

Несколько слов о каждом из этих инструментов.

- ANTLR[3] (ANother Tool for Language Recognition) — один из самых популярных сейчас генераторов. Главным его создателем является Теренс Пар (Terence Parr) из университета Сан-Франциско (США). Инструмент реализован на Java, но поддерживает в качестве целевых языков все наиболее распространенные сегодня языки программирования.
- ASF+SDF[4] (Algebraic Specification Formalism + Syntax Definition Formalism) — генератор с широкими возможностями, но достаточно сложным входным языком.
- Bison[5] — развитие инструмента YACC. Все грамматики, созданные для оригинального YACC, будут работать и в Bison.
- Coco/R[6](COmpiler COmpiler generating Recursive descent parsers) — академический проект, разрабатываемый в универ-

[1] http://dinosaur.compilertools.net.
[2] Имеется в виду огромное семейство инструментов, которые отличаются целевыми языками и незначительными модификациями алгоритма генерации.
[3] http://www.antlr.org.
[4] http://www.cwi.nl/projects/MetaEnv/.
[5] http://www.gnu.org/software/bison/.
[6] http://ssw.jku.at/Coco/.

ситете Линц (Австрия). Данный инструмент используется в Rotor (некоммерческой реализации платформы .NET) для создания компилятора и различных по назначению анализаторов языка C#.

- Elkhound[7] — генератор, который позиционируется как быстрый и удобный GLR-инструмент, созданный в университете Беркли (США).
- JavaCC[8] (Java Compiler Compiler) — популярный инструмент для создания синтаксических анализаторов на языке Java. Разрабатывается с 1996 г. при поддержке компании Sun Microsystems.
- Menhir[9] — реализация YACC с целевым языком Objective Caml, но имеющий значительно большую функциональность даже в сравнении с GNU Bison. Его создатели являются сотрудниками Французского национального института информатики и автоматизации (INRIA).
- SLK[10] (Strong LL(k)) — позиционируется как единственный настоящий $LL(k)$-генератор. Этот инструмент умеет преобразовывать грамматики (в нотации IEEE, ISO или YACC) в свой собственный формат, а также устранять из них левую рекурсию.

Обозначив круг рассматриваемых инструментов, перейдем к критериям их сравнения.

2. Критерии сравнения

Для сравнения инструментов автоматизации создания синтаксических анализаторов можно выделить ряд критериев. Во-первых, это качество инструмента как программного продукта [4]. Сюда входит и качество сопровождения продукта (регулярность выхода новых версий), и качество документации. При плохом сопровождении (или вообще его отсутствии) придется смириться с дефектами

[7] http://www.cs.berkeley.edu/~smcpeak/elkhound/.
[8] https://javacc.dev.java.net/.
[9] http://cristal.inria.fr/~fpottier/menhir/.
[10] http://home.earthlink.net/~slkpg/index.html.

Таблица 1. **Инструменты как программные продукты**

Название инструмента	Характеристики программного продукта	
	Сопровождаемость	Качество документации
ANTLR	3	3
ASF+SDF	2	1
Bison	3	2
Coco/R	2	3
Elkhound	3	1
JavaCC	3	2
Menhir	1	2
SLK	2	1

и недостатками генератора, если он является коммерческим или невозможно найти его исходные тексты. Если же инструмент имеет открытый исходный код, т. е. шанс самостоятельно исправить его дефекты в случае плохого сопровождения, хорошая документация позволит не только быстро изучить возможности генератора, но и применять эти средства наиболее эффективно.

В таблице 1 можно найти сравнение качества рассматриваемых инструментов, но только по двум характеристикам программного продукта — сопровождению и наличию хорошей документации. По другим метрикам очень трудно получить адекватные результаты сравнения, в то же время выбранные нами характеристики также не претендуют на абсолютную объективность. Каждому из генераторов выставлен некоторый балл (от 1 до 3) по каждой из метрик. Сопровождение оценивалось по количеству релизов, их доступности и регулярности выхода новых версий на момент написания статьи. Наивысший балл (3) здесь получили инструменты с богатой историей релизов и частым выпуском новых версий. Наименьший балл (1) оказался только у генератора Menhir с единственной доступной версией. Качество документации оценивалось с точки зрения ее полноты, удобства при проведении исследовательской деятельности и при ее использовании в качестве справочника. Инструменты, обладающие, по нашему мнению, полной, хорошо структурированной документацией, снабженной достаточным количеством примеров, получили по 3 балла. Не совсем пол-

Таблица 2. **Целевые языки инструментов**

Название инструмента	Языки программирования					
	C	C++	C#	Java	OCaml	Python
ANTLR	−	+	+	+	−	+
ASF+SDF	+	−	−	−	−	−
Bison	+	−	−	−	−	−
Coco/R	+	+	+	+	−	−
Elkhound	−	+	−	−	+	−
JavaCC	−	−	−	+	−	−
Menhir	−	−	−	−	+	−
SLK	+	+	+	+	−	−

ная документация (и/или не слишком удобная в качестве справочника) была оценена в 2 балла. Неполной, слишком краткой (и/или сложной) документации с малым числом примеров мы выставили 1 балл. Стоит отметить, что в коммерческом проекте нельзя оставить без внимания и условия лицензирования используемого инструмента.

Другим важным критерием является целевой язык генератора. Все инструменты принимают на вход описание трансляции, представленное в виде атрибутной грамматики [10] разбираемого языка, снабженной директивами инструмента. В результате работы генератора мы получаем один или несколько файлов, содержащих исходный код искомого анализатора (лексического, синтаксического или обоих) на одном из языков программирования (целевом языке). На каком именно языке будет создан анализатор, зависит от указанных опций и возможностей используемого инструмента. Какие языки поддерживаются конкретными генераторами можно узнать из таблицы 2.

Стоит отметить, что помимо указанных в таблице 2 существуют версии Coco/R для Delphi, Modula-2, Oberon и некоторых других языков, но они незначительно отличаются по своим возможностям и языку описания трансляций. Также для всех современных языков программирования общего назначения реализованы генераторы на основе оригинального YACC.

Несмотря на всю значимость перечисленных выше критериев, в процессе разработки спецификации (атрибутной грамматики) разбираемого языка наиболее важными оказываются такие возможности инструмента, как:

- класс языков, которые можно распознать при использовании данного генератора;
- средства описания грамматик и их повторного использования;
- методы разрешения конфликтов в грамматике;
- встроенные средства лексического анализа;
- механизм обработки ошибок разбора (в порождаемых анализаторах).

Эти возможности определяют практичность и удобство использования конкретного инструмента. Именно поэтому в настоящей статье мы проводим анализ современных генераторов по этим пунктам. Кроме того, не стоит забывать о технических деталях, посвоему важных при разработке конкретных программных продуктов. Например, при использовании LEX инструмент лишается возможности автоматической привязки лексем с точностью до колонки, что очень важно для точного позиционирования диагностических сообщений. Все же современные инструменты обладают такой функциональностью.

3. Применяемые алгоритмы

Внутренний алгоритм анализатора, порожденного каждым из расматриваемых инструментов, является вариантом либо LL, либо LR-разбора и определяет одну из характеристик этого инструмента — класс языков, принимаемый сгенерированными анализаторами.

При LL-алгоритме (его также называют *нисходящим анализом* [3]) дерево разбора строится от корня к листьям. Выбор между двумя (или более) альтернативами происходит в зависимости от первых лексем входной строки. Их количество, по которому определяется применяемое ко входной цепочке правило, обычно обозначают буквой k и указывают при наименовании алгоритма.

Существуют два способа реализации нисходящего анализа: табличный метод и рекурсивный спуск. Второй вариант достаточно прост, поэтому его часто используют при создании анализаторов

вручную. Табличная реализация LL-алгоритма применяется в SLK, а рекурсивный спуск — в ANTLR, Coco/R и JavaCC. При использовании LL-инструментов следует избегать левой рекурсии в правилах задаваемой грамматики, а вместо нее применять правую рекурсию.

LR-алгоритм (*восходящий анализ* [3]) строит дерево разбора от листьев к корню. Обычно число лексем входного потока, которые может просматривать LR-анализатор без изменения внутренней конфигурации (состояния), обозначают буквой k и указывают в скобках: $LR(k)$. Для хранения символов грамматики используется стек, а для анализируемой строки — входной буфер. Синтаксический анализатор работает путем *переноса* (`shift`) нуля или нескольких символов в стек до тех пор, пока на вершине стека не окажется некоторая подстрока входной строки (основа β). Затем он делает *свертку* (`reduce`) β к левой части соответствующей продукции. Другими словами, на каждом шаге алгоритма некоторая подстрока входной строки, соответствующая правой части какого-то правила (продукции), заменяется на левую часть этого правила. Анализатор повторяет этот цикл, пока не будет обнаружена ошибка или он не придет в конфигурацию, когда в стеке находится только стартовый символ, а входной буфер пуст. Таким образом, осуществляется свертка входной строки к стартовому символу грамматики.

LR-анализатор использует таблицу, которая состоит из двух частей: функции действий синтаксического анализа (`action`) и функции переходов (`goto`). Управляющая программа LR-анализатора определяет s_m (текущее состояние на стеке) и a_i (текущий входной символ). Затем программа обращается к ячейке таблицы действий синтаксического анализа, определяемой состоянием s_m и символом a_i, которая может иметь одно из четырех значений: перенос s (где s — это состояние), свертка в соответствии с продукцией $A \to \beta$, допуск (синтаксический анализ завершается), ошибка (обнаруживается ошибка и вызывается подпрограмма восстановления после нее). Функция `goto` получает в качестве аргументов состояние и символ грамматики и возвращает новое состояние. Данный алгоритм сложен для реализации вручную; работу LR-анализатора трудно отлаживать.

В силу того, что LR-анализаторы обладают большими таблицами, были изобретены $LALR$-анализаторы (*lookahead LR*). Они являются специальной формой LR-анализаторов и обладают значительно меньшими по размерам таблицами, чем у LR. К $LALR$-инструментам относится Bison.

При одинаковом k класс языков $LR(k)$ шире, чем класс $LL(k)$. LR–генераторами являются ASF+SDF, Elkhound и Menhir.

Для разрешения возникающих в LR-генераторах `shift-reduce` и `reduce-reduce` конфликтов (подробно рассматриваются в части 6 настоящей статьи) была изобретена некоторая надстройка над LR — обобщенный LR-алгоритм (GLR [8]). Он использует обычную таблицу LR-переходов, причем работает в точности, как LR-алгоритм до тех пор, пока не встретит конфликт. В этот момент GLR разветвляется и порождает столько анализаторов, сколько существует различных вариантов. Все эти анализаторы синхронизированы по операции `shift` и работают параллельно. Это позволяет объединять анализаторы, находящиеся в одинаковом состоянии. После работы GLR-алгоритма на выходе получается не одно дерево, как в случае LR-анализа, а набор деревьев (лес). Сократить их количество можно при помощи правила ассоциативности и приоритетов операций. Этот алгоритм используется в ASF+SDF, Elkhound и Bison (при установке опции `glr-parser`).

С момента создания первых генераторов прошло немало времени, быстродействие машин возросло, а объем их памяти существенно увеличился. Поэтому интерес к $LALR$ и LR-инструментам постепенно угасает из-за их сложной ручной реализации и неудобной отладки. На смену им приходят GLR-генераторы, которые более практичны, но так же сложны в отладке. Стоит отметить LL-инструменты, использующие рекурсивный спуск с простой реализацией и удобной отладкой.

4. Основные конструкции для описания грамматик

Для того чтобы получить анализатор требуемого языка, сначала нужно задать его атрибутную грамматику в обозначениях используемого инструмента. Хотя у каждого генератора свой мета-язык (некоторый контекстно-свободный язык [3, 7]), семантически все они достаточно похожи.

Синтаксическое правило состоит из левой и правой частей. В левой части находится название правила (нетерминальный символ), а в правой — цепочка (последовательность) терминальных и нетерминальных символов, в которые при разборе раскрывается

данное правило. Терминальными символами являются лексемы, а нетерминальными — названия синтаксических правил. Например, используя синтаксис мета-языка в ANTLR, присваивание в разбираемом языке можно задать так:

```
assignment: IDENT ASSIGN expr;
```

Согласно такому определению оператор присваивания (`assignment`) имеет следующий синтаксис: сначала идет идентификатор (`IDENT`), потом символ присваивания (`ASSIGN`), а затем следует некоторое выражение (`expr`). Символы `IDENT` и `ASSIGN` — терминалы (то есть лексемы), а `expr` — нетерминал, определяемый соответствующим синтаксическим правилом. Любое корректное синтаксическое правило должно раскрываться в последовательность лексем.

Для определения синтаксических правил во всех генераторах применяется форма Бэкуса-Наура (BNF [3]) или ее расширенный вариант (EBNF [3]), хотя в разном объеме и часто с различным синтаксисом.

В расширенной форме Бэкуса-Наура введены операции `*` и `+`. Запись `( rule )*` означает, что для разбора входного потока следует применить это правило несколько раз или же совсем его не использовать. В свою очередь, конструкция `( rule )+` обозначает повторение правила хотя бы один раз. Поэтому запись `( rule )+` эквивалентна записи `rule ( rule )*`. В большинстве инструментов (табл.) такие конструкции есть и имеют вид `( ... )*` или `{ ... }*` и `( ... )+`, или `{ ... }+`. Если же данные конструкции не поддерживаются генератором (YACC, Bison), то их не сложно выразить с использованием рекурсии и оператора «или» (`|`), который есть в любом инструменте. Помимо перечисленных выше, во многих генераторах существует конструкция вида `( ... )?` или `[ ... ]`, означающая опциональное применение правила ко входному потоку. Анализатор определяет, возможно ли разобрать входной поток в соответствии с этим правилом, и если возможно, то разбирает, иначе переходит к другим правилам. Такая конструкция может быть раскрыта как `( ... | )`. Описанные конструкции очень удобны в использовании и уменьшают объем создаваемой грамматики, тем самым повышая ее качество.

Во всех инструментах синтаксические правила можно снабдить *семантическими действиями* (кодом на целевом языке), поэтому

грамматика становится зависимой от конкретного языка программирования, что неудобно при последующем переходе на другой язык реализации. В некоторых генераторах (например, SLK) для этого предусмотрен дополнительный уровень абстракции: все семантические действия (семантики) помещены в отдельный файл, а ссылка на них из грамматики осуществляется по соответствующей метке (семантики снабжены метками). Таким образом, при переходе на другой язык реализации требуется только изменить файл с семантическими действиями, а файл с грамматикой останется без изменений. В то же время, такой подход не очень удобен: при разработке надо постоянно переключаться между двумя файлами, чтобы следить за соответствием семантик правилам грамматики, а при ее изменении не забывать исправить и семантические действия. Во всех современных инструментах можно задавать семантики в начале, середине или в конце правила.

Почти все генераторы (кроме Bison) позволяют именовать значения, возвращаемые правилами. В терминах атрибутных грамматик такие значения называются *синтезируемыми атрибутами*. Пример в обозначениях ANTLR:

```
expr returns [int e]:
   l=expr  '+'  r=expr { e = l + r; };
```

Таблица 3. **Конструкции для задания синтаксических правил**

Название инструмента	Повторения			Или
	0 или 1	≥ 0	≥ 1	
ANTLR	`(...)?`	`(...)*`	`(...)+`	`|`
ASF+SDF	`{...}?`	`{...}*`	`{...}+`	`|`
Bison	нет	нет	нет	`|`
Coco/R	`[...]`	`{...}`	нет	`|`
Elkhound	нет	нет	нет	`;`
JavaCC	`(...)?`	`(...)*`	`(...)+`	`|`
Menhir	`(...)?`	`(...)*`	`(...)+`	`|`
SLK	`[...]`	`{...}`	`{...}+`	`\n`

В YACC именования значений нет, а обращаться к ним можно только так:

```
expr: expr  '+'  expr { $$ = $1 + $3; };
```

При добавлении новых терминалов (нетерминалов) в определение правила `expr` все обращения к значениям правил, расположенных после вновь добавленных, станут недействительны, поскольку их нумерация сдвинется на единицу.

Основные конструкции и операции для задания синтаксических правил в инструментах представлены в таблице 3. Теперь рассмотрим расширенные средства различных генераторов, которые можно использовать при определении синтаксических правил.

Часто бывает удобно передать некоторые параметры в правила. Например, требуется создать анализатор языка, в котором процедуры (подпрограммы) описываются следующим образом:

```
SUBROUTINE A (/* формальные параметры */);
  BEGIN
    // операторы
  END A;
```

Как видно из примера, название процедуры указывается два раза. Нужно проверить, что идентификаторы одинаковы. В нотации ANTLR (семантические действия написаны на C++) решение выглядит так:

```
subroutine:
  // имя процедуры - идентификатор
  "SUBROUTINE" name:IDENT

  // формальные параметры
  parameters
  block[name->getText()];
block[const string & subroutineName]:
  "BEGIN" stmtList /* операторы */ "END" name:IDENT
  {
    if (subroutineName != name->getText())
      // обработка ошибки
  };
```

Помимо ANTLR передавать параметры в правила позволяют Coco/R и JavaCC. Если говорить в терминах атрибутной грамматики, то параметры являются наследуемыми атрибутами. Использование параметров не только удобно, но и позволяет влиять на

ход анализа входного потока, тем самым расширяя класс разбираемых языков (таким образом, можно распознавать класс языков, который больше, чем контекстно-свободные языки [3, 7]).

В ANTLR и Coco/R существует групповой символ («.» и «`ANY`» соответственно). Групповой символ обозначает любую лексему, например (в нотации ANTLR):

```
someRule : TOKEN_A . TOKEN_B;
```

В данном примере анализатор принимает любую одиночную лексему между `TOKEN_A` и `TOKEN_B`. В ANTLR, кроме того, поддерживается оператор отрицания («~»). Он применяется, когда требуется распознать любую лексему, за исключением какого-то их множества, например:

```
someRule : TOKEN_A ( ~TOKEN_B )* TOKEN_B;
```

В данном случае за `TOKEN_A` могут идти несколько произвольных лексем, кроме `TOKEN_B`, а затем идет `TOKEN_B`. Такие операторы есть еще в нескольких инструментах, но их можно использовать только при определении лексических правил (мы рассмотрим их позднее).

5. Лексический анализ

Лексический анализ [3] сопутствует синтаксическому разбору. На практике из-за сложности синтаксического анализатора его стараются упростить за счет использования возможностей лексического анализатора (сканера), поэтому мы уделяем лексическому анализу особое внимание.

Первые генераторы порождали только синтаксические анализаторы, а сканер разрабатывали вручную или использовали для его создания отдельную программу (например, LEX). Наиболее популярной среди современных инструментов для создания обособленных лексических анализаторов является программа FLEX[11] (The Fast Lexical Analyzer).

Большая часть современных инструментов (табл. 4) порождает лексические анализаторы вместе с синтаксическими, при этом возможна генерация исключительно сканера. При таком подходе,

[11] http://www.gnu.org/software/flex.

Таблица 4. **Возможности лексического анализа различных инструментов**

Название инструмента	Лексический анализатор		
	встроенный	внешний	тип алгоритма
ANTLR	+	−	$LL(k)$
ASF+SDF	+	−	
Bison	−	+	
Coco/R	+	+	DFA
Elkhound	−	+	
JavaCC	+	−	DFA
Menhir	−	+	
SLK	−	+	

с одной стороны, для создания лексического анализатора не требуется стороннего инструмента, который должен быть совместимым с данным генератором. С другой стороны, при описании лексических правил мы ограничены возможностями конкретного инструмента. Поэтому существуют генераторы, поддерживающие оба подхода, при которых можно как написать свой сканер, так и использовать внутренний (встроенный), что повышает их гибкость. Далее мы рассматриваем только встроенные возможности для создания сканеров в различных инструментах.

При определении спецификации лексического анализатора лексемы задаются с помощью лексических правил, которые имеют ту же структуру, что и синтаксические правила, поэтому при их описании можно использовать те же конструкции. Помимо них инструменты могут предоставлять дополнительные возможности (табл. 5). В первую очередь, это регулярные выражения [3], которые лучше всего поддержаны в JavaCC, где есть не только операции «*» и «+», но и возможность указать конкретное число повторений. Например, лексему, состоящую из семи букв, в JavaCC можно определить так:

```
TOKEN : { < #LETTER : ['a'-'z', 'A'-'Z'] > }
TOKEN : { < SEVEN_LETTER_WORD : (< LETTER >){7} > }
```

При задании `SEVEN_LETTER_WORD` мы воспользовались именованным подвыражением `LETTER`. Его можно использовать только как составную часть других лексем, поэтому такое подвыражение можно назвать «внутренней» лексемой, которая не будет возвращена

анализатору. Стоит отметить, что именованные подвыражения не могут ссылаться на себя рекурсивно (даже через некоторую цепочку подвыражений), поскольку в противном случае мы получим нерегулярную грамматику [3, 7].

Как видно из примера, признаком именованного подвыражения является символ «`#`». В ANTLR также есть понятие внутренней (защищенной) лексемы (для ее создания используется ключевое слово `protected`). В Coco/R именованные подвыражения описываются в специальной секции (`CHARACTERS`).

Возможности и качество кода порождаемого сканера зависят от алгоритма, в соответствии с которым он будет работать. В большинстве генераторов создаваемые лексические анализаторы работают по принципу детерминированного конечного автомата (DFA). ANTLR порождает сканеры с просмотром на k символов вперед ($LL(k)$-сканеры), реализованные по принципу рекурсивного спуска. Такой подход имеет ряд преимуществ. Во-первых, код сгенерированного лексического анализатора можно легко прочесть и отладить, как если бы он был написан вручную. Во-вторых, у такого сканера есть стек (в отличие от DFA), поэтому можно распознавать различные вложенные структуры, например, вложенные комментарии.

В Coco/R для компенсирования недостатков алгоритма DFA введены специальные средства. Так, для задания комментариев в разбираемом языке существует конструкция `COMMENTS`:

```
COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO eol
```

В примере определены два вида комментариев: в первой строке — многострочный вложенный комментарий, начинающийся символами «`/*`» и заканчивающийся «`*/`», а во второй — однострочный комментарий, начинающийся с «`//`» и продолжающийся до конца строки. Поскольку в Coco/R нет возможности просмотра вперед на несколько символов (в силу особенности внутреннего алгоритма), для создания контекстно-зависимых лексем предусмотрена дополнительная конструкция `CONTEXT`. Рассмотрим следующий пример:

```
integer = digit {digit} | digit {digit} CONTEXT ("..").
float = digit {digit} '.'
             {digit} ['E' ['+'|'-'] digit {digit}].
```

Таблица 5. **Расширенные операции для определения лексического анализатора**

Название инструмента	Диапазон	Групповой символ	Множественные операции			
			Отрицание	⋃	⋂	Разность
ANTLR	`..`	`.`	`~`	`|`	нет	нет
ASF+SDF	`-`	нет	`~`	`\/`	`/\`	`/`
Coco/R	`..`	`ANY`	нет	`+`	нет	`-`
JavaCC	`-`	`~[]`	`~`	`,`	нет	нет

Здесь конструкция `CONTEXT` позволяет лексическому анализатору различить число с плавающей точкой (к примеру, 2.39) и целое как начало диапазона чисел (например, 2..39), чего нельзя сделать при просмотре только одного символа вперед. С использованием данной конструкции сканер распознает оба случая корректно.

В таблице представлены дополнительные операции для задания лексических правил. Групповой символ используется для обозначения произвольного символа входного потока. Операции диапазон, отрицание, объединение (⋃), пересечение (⋂) и разность — соответствующие операции для работы с множествами символов. Например, в ASF+SDF лексему `LetterOrDigit` можно задать при помощи операций объединения и диапазона:

```
[a-z] \/ [A-Z] \/ [0-9] -> LetterOrDigit
```

Для определения однострочного комментария удобно использовать отрицание следующим образом (пример в нотации ANTLR):

```
COMMENTS : "//" ( ~'\n' )* '\n';
```

То же самое в Coco/R (где нет отрицания) можно описать так:

```
CHARACTERS newline = '\n'
           notNewline = ANY - newline
TOKENS comments = "//" { notNewline } newline
```

Теперь остановимся на дополнительных возможностях лексического анализа, которые не попали в таблицу.

В JavaCC есть понятие лексического состояния (lexical state), которое позволяет создавать лексему на протяжении нескольких лексических правил. По умолчанию определено только одно лексическое состояние `DEFAULT`, которое устанавливается в начале работы программы. Далее, помечая лексические правила состояниями (пользовательскими идентификаторами), можно добиться указанной функциональности, например:

```
<DEFAULT> MORE : { "h" : S1 }
<S1> MORE : {
  "e" {
    int l = image.length()-1;
    image.setCharAt(l, image.charAt(l).toUpperCase());
  } : S2
}
<S2> TOKEN : { "llo" { x = image; } : DEFAULT }
```

Находясь в состоянии `DEFAULT` и распознав во входном потоке символ «`h`», анализатор переходит в состояние `S1`. Далее, распознав «`e`» и преобразовав этот символ к верхнему регистру, сканер переходит в состояние `S2`. И, наконец, распознав «`llo`», лексический анализатор возвращается в состояние `DEFAULT` и выдает анализатору очередную лексему с текстом «`hEllo`». При этом модификатор `MORE` заставляет сканер продолжить разбирать входной поток, а слово `TOKEN` показывает анализатору, что обработка лексемы закончена.

В ASF+SDF можно создавать «запрещенные» продукции при помощи директивы `reject`:

```
lexical syntax
  [a-zA-Z][a-zA-Z0-9]* -> Id
context-free restrictions
  Id -/- [a-zA-Z0-9]
context-free syntax
  "begin" -> Id {reject}
```

В этом примере атрибут `reject` применяется для запрещения разбора зарезервированного слова `begin` как идентификатора (`Id`). Здесь используется также контекстно-свободное ограничение при определении идентификатора. Общий вид этой конструкции таков:

```
<Symbol>+ -/- <Lookaheads>
```

Под `<Symbol>` понимается правило, тип или литерал (в зависимости от разновидности ограничения), а под `<Lookaheads>` — некоторое множество символов. Смысл конструкции заключается в том, что во входном потоке за сущностями, указанными слева от оператора ограничения («-/-»), не могут следовать символы, записанные справа от оператора. Помимо контекстно-свободных ограничений в ASF+SDF существуют и лексические, например:

```
lexical restrictions
  "let" "in" -/- [a-z]
```

В ANTLR существует возможность фильтровать входной поток в соответствии с заданными лексическими правилами. Фильтр на входной поток указывается в секции опций в виде `filter = <правило>`, а само правило должно быть помечено как защищенное (`protected`). Но можно задать фильтр и проще, написав `filter = true`. Тогда анализатор пропустит во входном потоке все символы, которые невозможно разобрать с помощью имеющихся лексических правил, например:

```
class T extends Lexer;
options {
  k = 2;
  filter = true;
}
P : "<p>"  { /* семантические действия */ };
BR: "<br>" { /* семантические действия */ };
```

Таким образом, мы «извлечем» из входного потока (html–файла) все теги вида `<p>` и `<br>`. Возможность пропускать (игнорировать) некоторые конструкции во входном потоке удобна на практике, например, она позволяет легко отсеять комментарии или директивы препроцессора, или и то, и другое.

6. Разрешение неоднозначностей в грамматике

На практике разбираемый язык часто имеет неоднозначную грамматику, т. е. такую грамматику, которая дает более одного дерева вывода для некоторой строки лексем. Другими словами, генератор не может для какого-то правила грамматики создать анализатор, делающий корректный выбор между двумя или более продукциями. Отметим, что тип неоднозначности зависит от алгоритма, применяемого в инструменте. LR-инструменты для неоднозначной грамматики могут достичь конфигурации, в которой по информации о содержимом своего стека и об очередном входном символе не в состоянии решить, должен ли использоваться перенос или свертка (*shift-reduce конфликт*), либо какая из нескольких возможных сверток должна применяться (*reduce-reduce конфликт*). Алгоритм GLR, в отличие от LR, никогда не принимает такого решения, а разбирает все возможные варианты, строит лес синтаксического разбора и затем уменьшает его при помощи некоторых семантических правил.

Типичным примером shift-reduce конфликта служит оператор «`if-then-else`»:

```
if_stmt: "if" expr "then" stmt;
if_else: "if" expr "then" stmt
         "else" stmt;
```

В качестве примера reduce-reduce конфликта приведем следующие два правила:

```
expr: expr "+" expr;
prefix_plus: "+" expr;
```

В силу специфики своего алгоритма, LL-генераторы (в отличие от LR) должны сразу выбрать одно из правил, по которому будет происходить разбор входной цепочки. Этот выбор осуществляется по первым лексемам входной строки. Может возникнуть ситуация, когда встречаются правила с одинаковым префиксом и не понятно, какое из них применить. По аналогии с shift-reduce и reduce-reduce конфликтами назовем эту ситуацию *shift-shift конфликтом*. Примером ее служит все тот же оператор «if-then-else».

Все современные инструменты предлагают те или иные средства для разрешения неоднозначности. Помимо внутренних механизмов генератора, на которые никак нельзя повлиять, существуют средства, которыми можно управлять в той или иной степени. Выделим два вида таких средств: директивы инструментов (опции, атрибуты — как на уровне всей грамматики, так и на уровне отдельных правил) и специальные конструкции метаязыка — резольверы [7].

В дальнейшем мы будем называть их также предикатами, так как на практике часто используют именно этот термин.

Во всех *LR*-генераторах (ASF+SDF, Bison, Elkhound, Menhir) есть директивы `left`, `right`, `non-assoc` (или `nonassoc`). С помощью них для операторов задается предшествование и ассоциативность, которые помогают избежать shift-reduce конфликта. Опция `left` означает левую ассоциативность, т. е. если `op` — операция, то запись `x op y op z` эквивалентна записи `(x op y) op z`. Аналогично, директива `right` используется для задания правой ассоциативности (выражение `x op y op z` в этом случае будет эквивалентно выражению `x op (y op z)`). Если операция `op` не является ассоциативной, точнее, если к ней нельзя применить ни правило левой, ни правило правой ассоциативности, то следует использовать директиву `non-assoc`, тогда запись вида `x op y op z` будет вызывать синтаксическую ошибку. В нотации Bison приоритет (предшествование) операции определяется порядком записи. В начале записываются операции с более низким приоритетом, а с одинаковым — на одной строке. Например, в нотации Bison описание операций «`-`», «`+`», «`*`», «`/`», «`|`» унарный минус и возведение в степень будет выглядеть так:

```
%left '-' '+'
%left '*' '/'
%left UMINUS  /* унарный минус */
%right '^' /* возведение в степень */
```

Самый низкий приоритет по сравнению с другими операциями у сложения и вычитания, поэтому они записаны в начале. Поскольку и сложение, и вычитание имеют равный приоритет (и одновременно левоассоциативны), они находятся на одной строке. Самый высокий приоритет у операции возведения в степень, поэтому она записана в самом конце. Если не указать ассоциативность и приоритет, то грамматика окажется неоднозначной. Например, при разборе выражения `2 + 3 * 9` было бы не понятно, как его сгруппировать: `(2 + 3) * 9` или `2 + (3 * 9)`. В следующем примере используется еще одна директива (`prec`):

```
expr: ...
   | expr '-' expr
   ...
   | '-' expr \%prec UMINUS
```


Директива `prec` задает выражению `'-' expr` такой же приоритет, как и у фиктивной лексемы `UMINUS`. Благодаря этому унарный минус будет правильно распознан во входном выражении и reduce-reduce конфликт удачно разрешится. Данная опция помимо Bison есть в Elkhound и Menhir.

В некоторых инструментах (ASF+SDF, Elkhound и Bison при установке опции `glr-parser`) применяется *GLR*-алгоритм [8]. Как уже говорилось, он разбирает все возможные варианты и строит лес. Если в конечном итоге необходимо получить одно синтаксическое дерево, то придется применить фильтр для леса. Разрешение конфликтов в базовом *LR*-анализаторе уменьшает число деревьев при *GLR*-разборе. Кроме того, для него существуют свои директивы. Например, в Bison для явного указания выбора между альтернативами используется опция `dprec`. Рассмотрим ситуацию, в которой некоторые входные цепочки могут быть распознаны и как `expr`, и как `decl`:

```
stmt : expr ';' \%dprec 1
     | decl     \%dprec 2;
```

В случае корректного разбора входного потока по этим двум правилам *GLR* использует `expr` и построит одно дерево, поскольку приоритет первого правила, указанный с помощью опции `dprec`, выше, чем у второго. Директиву `merge` следует использовать, когда необходимо рассмотреть все возможности, а не выбрать какую–то одну из них. Для этого определение правила `stmt` должно выглядеть так:

```
stmt : expr ';' \%merge <stmtMerge>
     | decl     \%merge <stmtMerge>;
```

Следует также задать функцию (в примере она называется `stmtMerge`), которая будет обрабатывать случай неоднозначности, т. е. определять некоторые семантические действия, которые будут выполняться, когда анализатор не сможет сделать выбор между несколькими альтернативами (в данном случае — между `expr` и `decl`). В Elkhound директивы `merge` нет, но есть функция `merge`, которая делает то же самое, что и `stmtMerge`. В ASF+SDF для тех же целей служат другие директивы (атрибуты): `prefer, avoid` и `reject`. Первая опция является своеобразным аналогом директивы

`dprec` из Bison. Правило, помеченное атрибутом `avoid`, при неоднозначности не рассматривается, тем самым выбор альтернативы сужается. Директива `reject` означает, что правило не будет использовано для разбора входного потока вне зависимости от наличия неоднозначностей (в отличие от `avoid`).

LL-инструменты обладают своими средствами для решения неоднозначностей. Многие из них (ANTLR, JavaCC, SLK) предоставляют возможность задавать число лексем входного потока, на которое генератор может заглядывать (`lookahead`) при выборе одной из альтернатив. Этот способ помогает инструменту разрешить shift-shift конфликт. В ANTLR и SLK данная опция обозначается через k, а в JavaCC — через `LOOKAHEAD`.

Кроме опций в таких инструментах, как ANTLR, JavaCC и Coco/R, существуют специальные конструкции — синтаксические и семантические предикаты. Это еще один способ разрешения shift-shift конфликта. Например, в JavaCC синтаксические предикаты имеют вид:

```
LOOKAHEAD( rule1 ) rule2 |. rule3
```

что означает следующее: попытаться разобрать входной поток по правилу `rule1` и в случае успеха вызвать правило `rule2`, а при неудачном завершении `rule1` выполнить правило `rule3`. Синтаксические предикаты также поддержаны в ANTLR, а в Coco/R приходится вручную реализовывать их некоторый аналог.

Семантические предикаты в нотации ANTLR выглядят так:

```
{ <выражение> }?
```

Если `выражение` принимает истинное значение, то правило (внутри которого встретился этот семантический предикат) продолжит свое выполнение в обычном режиме, в противном случае оно тут же закончит свою работу и передаст управление в вызвавший его метод. Семантические предикаты также реализованы в JavaCC и Coco/R.

Обобщим вышесказанное. Тип неоднозначности зависит от алгоритма, применяемого в инструменте. В *LR*-генераторах возникают shift-reduce и reduce-reduce конфликты, *GLR* разбирает все возможные варианты и строит лес, а в *LL* имеет место shift-shift конфликт. *LR*-инструменты разрешают свои конфликты при помощи специальных атрибутов и директив: предшествование и ассоциативность операторов или явное указание выбора между альтернативами в случае неоднозначности. В *GLR* количество деревьев разбора сокращается при помощи правила ассоциативности и приоритетов операций. Для разрешения конфликтов в *LL*-инструменте задается число лексем входного потока, на которое генератор может заглядывать при выборе одной из альтернатив, или используются синтаксические и семантические предикаты.

Несмотря на то что резольверы являются практичным средством решения конфликтов, их чрезмерное использование снижает скорость работы анализатора. В качестве примера рассмотрим синтаксический предикат, имеющий вид:

```
(rule1) => rule2 | rule3
```

Правило `rule1` может состоять из

```
(rule4) => rule5 | rule6
```

где `rule4`, в свою очередь, может иметь вид:

```
(rule7) => rule8 | rule9
```

и так далее. Таким образом, разбор первого правила будет иметь экспоненциальную сложность.

7. Средства повторного использования грамматик

Многие языки имеют схожий синтаксис, поэтому становится возможным повторное использование каких-то частей грамматики одного языка при создании анализатора другого языка. В рассматриваемых инструментах реализованы различные механизмы: описание грамматики в нескольких файлах (модульность), параметризация одних правил другими и наследование грамматик.

Разделение спецификации разбираемого языка на несколько файлов (модулей) не только позволяет повторно использовать какие-то ее части, но и упрощает процесс разработки и сопровождения, поскольку повышается ее управляемость и снижается сложность [5]. Поэтому кажется логичным, что ASF+SDF «знаком» только с понятием модуля. Каждый модуль состоит из нескольких секций. Сначала перечисляются импортируемые (используемые) модули (секция `imports`), далее следуют секции с описанием лексических (секция `lexical syntax`) и синтаксических правил

(секция `context-free syntax`). Каждая из таких секций помечена как `exports` (доступная из других модулей) или как `hiddens` (скрытая, недоступная из других модулей). Эти секции находятся в файле `<ModuleName>.sdf`, а последняя секция `equations` (в ней определяются функции, описанные в лексических и синтаксических правилах) располагается в файле `<ModuleName>.asf`. Таким образом, физически один модуль занимает два файла. Стоит отметить, что в ASF+SDF модульная структура входной спецификации поддержана лучше, чем в других инструментах.

В Menhir схожая функциональность реализована по-другому. Если на вход инструменту подается несколько файлов, то он автоматически «склеивает» их в один с целью получения единой спецификации. При этом склейка не является только текстовой. Разделяют общедоступные правила (такое правило должно быть помечено как `%public` или как стартовый символ — `%start`) и закрытые правила. К общедоступным правилам можно обращаться из других модулей, а к закрытым — нет. При объединении модулей закрытые правила с одинаковыми именами переименовываются (для предотвращения коллизий), а общедоступные правила (с одинаковыми именами) объединяются с помощью оператора «или» («`|`»).

В Elkhound существует директива `include`, которая обеспечивает подключение файлов грамматики, за счет чего достигается модульность входной спецификации. Как показывает практика, разделение грамматики разбираемого языка на части (модули) очень удобно. Тем не менее в Bison, Coco/R и JavaCC такая возможность отсутствует.

Генератор SLK требует, чтобы входная спецификация была разбита на фиксированное число файлов. Среди них должны быть файлы с грамматикой, семантическими действиями, лексическим анализатором и еще несколько вспомогательных файлов.

ANTLR не поддерживает модульность явно. Можно только разделить анализатор и сканер (описывать их в разных файлах). Кроме того, и анализатор, и сканер можно разбить на несколько файлов за счет наследования грамматик. Наследование грамматик позволяет расширять их за счет новых правил, а также перегружать старые правила (правила из базовой грамматики):

```
class Simple extends Parser; // базовая грамматика
  stat: ( expr ASSIGN expr ) | SEMICOLON;
  expr: ID;
```



```
class Derived extends Simple; // производная грамматика
  expr: ID | INT;
  ifStat: "if" expr "then" stat ( "else" stat )?;
```

Грамматика `Derived` (на самом деле это название класса анализатора, который сгенерирует ANTLR по данной спецификации) состоит из трех правил. Первое правило (`stat`) унаследовано из грамматики `Simple`, второе (`expr`) замещает соответствующее правило базовой грамматики, а третье правило (`ifStat`) является новым (и никак не связано с `Simple`). Такое наследование грамматик в стиле объектно-ориентированного программирования реализовано только в ANTLR, несмотря на то, что многие инструменты порождают анализаторы на объектно-ориентированных языках.

Обычно лексические и синтаксические правила отделяются друг от друга явно, точнее, описываются в различных секциях (частях) грамматики, хотя в некоторых случаях, например в JavaCC, описания лексических и синтаксических правил чередуются друг с другом. При этом вся спецификация разбираемого языка в JavaCC (впрочем, как и в Coco/R) оказывается в одном файле, который может достигать достаточно больших размеров даже для простых (с точки зрения конфликтов в грамматике) языков, содержащих достаточное число различных конструкций и команд, что представляется не очень удобным как для разработки, так и для сопровождения. Тем не менее, разработчики JavaCC считают это достоинством своего инструмента.

Еще одним средством повторного использования является параметризация правил с помощью других правил, которая поддержана только в ASF+SDF и Menhir. Например, в Menhir это может выглядеть так:

```
procedure(list): | PROCEDURE ID list(formal)
                   SEMICOLON block SEMICOLON { ... }
```

В данном случае правило `list` параметризует правило `procedure`. К примеру, список (`list`) может быть пустым

```
list(x) : | { пустой список }
          | list(x); COMMA; x
```

или же всегда состоять как минимум из одного элемента

```
list(x): | x
         | list(x); COMMA; x
```

Такая грамматика очень похожа на двухуровневую грамматику Вейнгаардена [11]. При этом правила `procedure`, `list` (из примеров) в терминах Вейнгаардена называются гиперправилами, а правило `x` — метаправилом. Данный формализм был использован при описании синтаксиса и семантики языка Algol68 [1].

В то время как в языках программирования средствам повторного использования кода давно уделяется достаточное внимание, в инструментах для создания анализаторов средства повторного использования грамматик, как мы видим, развиты слабо. О причинах такой ситуации можно только догадываться. Скорее всего это объясняется некоторым отставанием отрасли генераторов от развития языков программирования, поскольку эти инструменты имеют не столь широкую область применения.

8. Восстановление после ошибок

Восстановлением называют такое поведение анализатора в случае синтаксической ошибки, при котором он не только выдает сообщение о ней, но и продолжает разбор входного потока. Иногда к обработке ошибок при синтаксическом анализе выдвигаются повышенные требования, поэтому становится важным, какие средства восстановления предоставляются инструментом. Удобнее использовать стандартные механизмы, чем реализовывать их вручную, хотя и не во всех инструментах можно задать свою стратегию восстановления.

В ANTLR и JavaCC обработка ошибок осуществляются при помощи исключений. Кроме обработчиков исключений по умолчанию можно определить и свои. В JavaCC по умолчанию обработчик выдает ошибку и анализатор заканчивает свою работу. В ANTLR обработчик исключения также выдает ошибку, но анализатор продолжает работу, пропуская лексемы до первой подходящей, с которой можно продолжить разбор входного потока. Тем самым в ANTLR не приходится реализовывать собственные обработчики исключений, если к диагностике ошибок не выдвигаются повышенные требования.

JavaCC предлагает два вида восстановления: *поверхностное* (*shallow recovery*) и *глубокое* (*deep recovery*). Первое из них имеет

место, когда для правила нельзя применить ни одну из его альтернатив, а второе — когда возникает ошибка внутри одной из них. Рассмотрим поверхностное восстановление:

```
void Stmt():
{}
{ IfStmt() | WhileStmt() | error_skipto(SEMICOLON) }

JAVACODE
void error_skipto(int kind)
{
  // выдаем сообщение об ошибке
  // перемещаемся во входном потоке до лексемы
  // с типом kind
}
```

Если при использовании правила `Stmt` для разбора входного потока нельзя применить ни одну из его альтернатив, то будет выдана ошибка, анализатор пропустит все лексемы до символа «`;`» и продолжит свою работу. В случае глубокого восстановления для правила `Stmt` следует писать:

```
void Stmt():
{}
{
  try {
    ( IfStmt() | WhileStmt() )
  }
  catch (ParseException e) {
    // выдаем сообщение об ошибке
    // перемещаемся во входном потоке до ";"
  }
}
```

Если внутри правила `IfStmt` или `WhileStmt` возникнет ошибка, которая там не обрабатывается, то синтаксический анализатор выдаст сообщение об ошибке и продолжит разбор, пропустив лексемы до «`;`».

Bison и Menhir поддерживают восстановление по умолчанию. В случае синтаксической ошибки они генерируют специальную лексему (`error`). Для задания своей стратегии восстановления можно

определить правило, обрабатывающее эту лексему в текущем контексте. В случае Bison это можно сделать следующим образом:

```
stmts:
| stmts ';'
| stmts exp ';'
| stmts error ';'
```

В этом примере в случае ошибки внутри правила `exp` синтаксический анализатор пропустит все лексемы до «`;`» и продолжит разбор.

Инструмент SLK имеет три стратегии для восстановления после ошибок. Эти стратегии реализованы как методы класса SlkError. Первая (метод `mismatch`) обрабатывает ситуацию, когда текущая лексема входного потока не совпадает с одной из ожидаемых. При этом выдается ошибка, и парсер пытается продолжить разбор с текущей лексемы. Вторая стратегия (метод `no_entry`) рассматривает случай, в котором с текущей лексемы не начинается ни одно из правил. Тогда анализатор выдает сообщение об ошибке и пропускает эту лексему. Последняя стратегия (метод `input_left`) применяется для ситуации, когда анализатор закончил разбор, а входной поток еще остался (по умолчанию в этом случае ничего не делается).

В Coco/R для восстановления после ошибок используются *точки синхронизации* (*synchronization points*), которые должны быть явно указаны в грамматике. Точки синхронизации определяются при помощи конструкции `SYNC`. В случае ошибки синтаксический анализатор восстанавливается в этой точке, пропуская все лексемы и выдавая ошибки, пока не встретит ожидаемую лексему. Часто такой точкой могут выступать начало команды (`if`, `while`, `for` и т. д.), области определения (`public`, `private`, `void` и т. д.) и разделитель команд (например, «`;`»). Символ конца файла всегда включается в набор точек синхронизации, что гарантирует корректное поведение анализатора при восстановлении. Рассмотрим следующий вариант использования точек синхронизации:

```
Stmt =
 SYNC
  ( Ident '=' Expr SYNC ';'
  | "while" '(' Expr ')' Stmt
  | ...
  ).
```


Если при использовании правила `Stmt` для разбора входного потока нельзя применить ни одну из его альтернатив, то анализатор пропустит лексемы, выдавая ошибки, пока не встретит подходящую (`Ident`, `while`, ...) и продолжит дальнейший разбор файла. Это похоже на поверхностное восстановление в JavaCC. В случае ошибки внутри правила `Expr` анализатор пропустит лексемы до «`;`», выдавая соответствующие сообщения. Это аналогично глубокому восстановлению в JavaCC.

Помимо точек синхронизации в Coco/R есть еще конструкция `WEAK`, позволяющая обрабатывать ситуацию, когда лексема пропущена в исходном файле или вместо нее используется другая. Такая ошибка часто встречается в различных перечислениях, например списке параметров:

```
ParameterList = '(' Parameter { WEAK ',' Parameter } ')'.
```

В этом случае если запятая будет пропущена или вместо нее будет использована другая лексема (например, «`;`»), то синтаксический анализатор выдаст сообщение об ошибке и продолжит обработку очередного параметра.

Без использования конструкций `SYNC` и `WEAK` анализатор выдаст сообщение об ошибке и закончит работу.

Заключение

В настоящее время нет инструмента для создания анализаторов, который превосходил бы другие генераторы по всем показателям. Можно выделить лидеров только по отдельным характеристикам.

На практике наиболее удобны инструменты, использующие LL-алгоритм с резольверами или GLR. Среди LL-генераторов предпочтительней других выглядят ANTLR, JavaCC и Coco/R. Все три инструмента порождают анализаторы, реализованные по методу рекурсивного спуска, поэтому сгенерированный код легко прочесть и отладить, в отличие от кода, порождаемого LR-генераторами. На наш взгляд, в этой тройке лидером по своим возможностям является ANTLR. Среди GLR-инструментов (Elkhound, ASF+SDF, Bison) мы можем выделить Elkhound, поскольку ASF+SDF имеет сложный входной язык, а в Bison GLR-алгоритм реализован неэффективно.

Наиболее удобными встроенными возможностями лексического анализа обладают ANTLR и JavaCC. Немного им проигрывают ASF+SDF и Coco/R.

Лучшие средства повторного использования предоставляет ASF+SDF, где модульность грамматики реализована на уровне спецификации входного языка. Неплохими средствами обладают Elkhound и Menhir, где возможно хотя бы разделить грамматику на достаточное число файлов, в отличие от ANTLR (где можно разбить грамматику только на два файла), JavaCC и Coco/R. В двух последних инструментах вся спецификация грамматики располагается в одном файле.

Список литературы

[1] Алгол 68. Методы реализации / Под ред. Г. С. Цейтина. Л.: Изд-во ЛГУ, 1976. 224 с.

[2] *Ахо А., Ульман Дж.* Теория синтаксического анализа, перевода и компиляции. Т. 1; Синтаксический анализ. 612 с. Т. 2; Компиляция. 487 с. М.: Мир, 1978.

[3] *Ахо А., Сети Р., Ульман Дж.* Компиляторы: принципы, технологии, инструменты. М.: Вильямс, 2003. 768 с.

[4] *Брукс Ф.* Мифический человеко-месяц или как создаются программные системы. СПб.: Символ-Плюс, 2001. 304 с.

[5] *Макконнелл С.* Совершенный код. СПб.: Питер, 2005. 896 с.

[6] *Мартыненко Б. К.* Синтаксически управляемая обработка данных. СПб.: Изд-во СПбГУ, 1997. 363 с.

[7] *Мартыненко Б. К.* Языки и трансляции. СПб.: Изд-во СПбГУ, 2002. 229 с.

[8] *Tomita M.* An Efficient Augmented-Context-Free Parsing Algorithm // Computational Linguistics, 13(1–2). 1987. P. 31–46.

[9] *Johnson S. C.* YACC — yet another compiler-compiler // Bell Telephone Laboratories, Computing Science Technical Report N 32. 1975.

[10] *Knuth D.* Semantics of Context-free Languages // Mathematical Systems Theory. Vol. 2. N 2. 1968. P. 127–145.

[11] *van Wijngaarden A.* Revised Report on the Algorithmic Language ALGOL 68 // Mathematisch Centrum. 1976. 236 p.

СОДЕРЖАНИЕ