Построение ослабленного LALR-транслятора на основе анализа грамматики на избыточность

А. А. Ефимов
Andrej.Efimov@gmail.com

Я. А. Кириленко
jake@math.spbu.edu

Данная статья посвящена задаче автоматического построения LALR-транслятора с функцией исправления ошибок определенного типа. Был выбран специальный класс ошибок, связанных, как правило, с отсутствием в тексте транслируемой программы обязательных ключевых слов, пропущенные вхождения которых тем не менее могут быть однозначно вычислены автоматически. Актуальность именно такого ограничения на класс обрабатываемых ошибок связана со спецификой реализации трансляторов в задачах реинжиниринга программного обеспечения: устаревшие компиляторы зачастую разрешали пропускать «очевидные», в определенном смысле слова, ключевые слова. Таким образом, большие объемы актуальных текстов программ, корректные с точки зрения пользователя и некорректные с точки зрения спецификации соответствующих языков, не могут быть обработаны средством автоматизированного реинжиниринга.

Введение

Зачастую устаревшие компиляторы разрешали пропускать в тексте программы «очевидные» в определенном смысле ключевые слова, заявленные в спецификации языка как «обязательные», и даже целые конструкции. При разработке средства автоматизированного реинжиниринга устаревших программ [4] такое отклонение

поведения оригинального компилятора от заявленного в спецификации приводило к неспособности разрабатываемого инструмента анализировать исходные тексты, корректные с точки зрения как оригинального компилятора, так и конечного заказчика. Стоит отметить также, что никакое тестирование, основанное на спецификации языка, не способно выявить такие ошибки заранее. Постоянно требовалась доработка грамматики трансляторов на основании жалоб заказчиков по примерам исходных текстов, подлежащих анализу. Поскольку встроенные средства реакции на ошибку в генераторах парсеров, используемых при создании упомянутого средства реинжиниринга, не решают эту задачу, авторами статьи была предпринята попытка предупреждения описанных проблем с помощью раннего выявления в языке «бесполезных» терминальных символов.

Искомое решение должно было обладать следующими важными свойствами. Во-первых, оно должно сохранить неизменной трансляцию корректных входных цепочек. Во-вторых, для любой некорректной цепочки[1], которую сможет принять модифицированный транслятор, результат должен совпадать с результатом трансляции некоторой корректной цепочки[2]. В-третьих, крайне важно, чтобы не требовалась модификация грамматики и построение «ослабленного» транслятора выполнялось автоматически. В-четвёртых, для простоты внедрения алгоритм должен быть ориентирован на LALR-трансляторы.

В данной статье предлгается алгоритм создания такого LALR-анализатора, который способен автоматически исправлять определенный класс ошибок (главным образом, пропущенные ключевые слова, отсутствие которых в тексте программы может быть определено автоматически) и опускать конструкции, не влияющие на результат. Алгоритм реализован на основе генератора парсеров Bison. В статье приводятся и разбираются результаты апробации данного алгоритма на примере грамматики языка программирования ANSI C.

[1]Здесь и далее под некорректной цепочкой будем подразумевать цепочку, которую не принимает оригинальный транслятор.

[2]Транслятор, обладающий этими свойствами, будем называть ослабленной модификацией оригинального транслятора, или ослабленным транслятором. Трансляцию, реализуемую ослабленным транслятором, будем называть ослабленной.

1. Обсуждение проблемы

По своей сути, построение «ослабленного» анализатора в данном случае сводится к задаче обработки некорректных цепочек. Такие задачи решают с помощью алгоритмов реакции на ошибку. Различают 3 основных способа реакции на ошибку:

1) обнаружение ошибок (*error detection*);
2) восстановление после ошибок (*error recovery*);
3) исправление ошибок (*error correction*).

Обнаружение ошибок — наименьшее, что можно потребовать от транслятора. Он лишь должен показать, что во входной строке есть синтаксическая ошибка (т. е. строка не принадлежит языку, описанному данной грамматикой).

В некоторых случаях обнаружения первой ошибки достаточно. Но, как правило, нужно знать обо всех ошибках во входной строке. В этом случае транслятор должен суметь продолжить работу, изменив свое внутреннее состояние. Такая обработка называется *восстановлением после ошибок*. В общем случае компилятор указывает позицию, в которой обнаружена ошибка, а также некоторое диагностическое пояснение, например «в данной позиции ожидается ';'».

Главный недостаток анализатора с восстановлением после ошибок — невозможность построить дерево синтаксического разбора при наличии ошибок во входной строке. При обнаружении ошибок изменение внутреннего состояния может привести к выполнению семантических действий, связанных с правилом грамматики, в том порядке, какой был бы невозможен при синтаксически корректном входе. Вследствие этого могут появиться наведенные ошибки.

Одним из вариантов решения данной проблемы является игнорирование семантических действий в том случае, если обнаружена ошибка. Но это не всегда приемлемо.

От средства реинжиниринга требуется уметь получать правдоподобную трансляцию при синтаксически некорректном входе. Например, одним из механизмов, который используется при оценке сложности проекта по реинжинирингу устаревшего ПО, является вычисление формальных метрик его исходных кодов (инвентаризация)[3]. В этом случае достаточно дать хотя бы приблизительный результат трансляции для некорректной входной цепочки.

[3] О различных метриках инвентаризации можно узнать здесь: http://www.stsc.hill.af.mil/crosstalk/1994/12/

Более подходящим вариантом является использование различных методов *исправления ошибок* [9, 10]. При исправлении ошибок анализатор преобразует входную строку в синтаксически корректную, удаляя, добавляя или изменяя терминальные символы. Стоит заметить, что такое преобразование не всегда формирует входную строчку, какая подразумевалась пользователем. Поэтому исправлению подлежат, как правило, только простые ошибки [7], например, отсутствие точки с запятой в конце конструкции.

Ввиду вышесказанного для поставленной цели наиболее удачным будет использование алгоритма исправления ошибок. Ниже мы рассмотрим классификацию таких алгоритмов.

Большинство современных широко распространённых генераторов парсеров реализуют LR-, LALR-, GLR-алгоритмы разбора [5]. В настоящей работе решение поставленной задачи будет рассмотрено на примере LALR-трансляторов [6, 10]. В открытом доступе существует большое количество LALR-грамматик языков программирования, что существенно упрощает анализ актуальности предлагаемого решения.

2. Обзор методов исправления ошибок

Сформулируем, какими свойствами должен обладать искомый алгоритм:

1) алгоритм должен сохранить неизменной трансляцию корректных входных цепочек;
2) для любой некорректной цепочки, которую сможет принять ослабленный транслятор, результат должен совпадать с результатом трансляции некоторой корректной цепочки;
3) входная грамматика должна оставаться неизменной, алгоритм должен автоматически строить ослабленный транслятор;
4) алгоритм должен быть ориентирован на табличный LALR-транслятор.

Рассмотрим основные классы существующих алгоритмов исправления ошибок.

2.1. Специальные методы

Методы данной группы ориентированы на определенные грамматики и не являются языковонезависимыми.

2.1.1. Метод пустых ячеек

Управляющая таблица LALR-транслятора содержит пустые ячейки, при обращении к которым обнаруживается ошибка. В этом алгоритме таким ячейкам сопоставляются процедуры обработки ошибок, написанные программистом. С одной стороны, этот алгоритм не зависит от грамматики, и его достаточно просто применить на практике. С другой стороны, он обладает существенным недостатком — для каждой пустой ячейки программист должен реализовать отдельную процедуру для обработки ошибки. Конечно, некоторые процедуры обработки могут использоваться для многих ячеек, тем не менее необходимо приложить немало усилий для анализа всей управляющей таблицы на возможные ошибки и внедрение процедур их обработки. Одна из реализаций данного алгоритма рассмотрена в работе [8].

2.1.2. Алгоритм добавления продукций ошибок

Данный алгоритм был предложен для того, чтобы снизить затраты на разработку анализатора и избавиться от некоторых недостатков предыдущего алгоритма. Алгоритм основан на расширении контекстно-свободной грамматики с помощью специальных продукций, называемых продукциями ошибок. Продукции ошибок — это правила грамматики, добавляемые программистом для того, чтобы возможные ошибки стали частью языка, т. е. перестали быть синтаксическими ошибками грамматики. Такие продукции, как правило, содержат семантическое действие, которое сообщает об ошибке. Программист должен заранее определить возможные ошибки ввода и способ их исправления. Несмотря на то, что этот алгоритм является достаточно общим, независимость от грамматики в нем также не была достигнута.

2.1.3. Метод добавления терминалов ошибки

Терминал ошибки — это специальный терминальный символ, добавляемый перед местом обнаружения ошибки (error detection point). При переносе данного терминала анализатор удаляет состояния из стека до тех пор, пока не будет принят этот терминал, а затем пропускает все символы из входной строки, пока не будет найден допустимый. В этом алгоритме также необходимо расширить грамматику новыми правилами, содержащими терминал ошибки.

2.2. Коррекция на уровне фраз

Алгоритмы этого класса обычно состоят из двух этапов. Первый, называемый фазой сгущения, накапливает информацию об участке входной цепочки вокруг ошибки. Второй, называемый фазой исправления, возвращает сообщение, содержащее информацию об ошибке, и указывает способ ее исправления.

На первом этапе анализатор пытается разобрать непрочитанную часть входной строки, не обращая внимания на контекст слева, но накапливая контекст справа. Анализ продолжается до тех пор, пока не будет сделана попытка свернуть ошибочный терминал или не встретится новая ошибка. Таким образом, подобный анализ может быть рекурсивным.

На втором этапе анализатор модифицирует входную цепочку или стек, используя информацию, собранную на первом этапе. Как правило, различным модификациям ошибок присваиваются разные стоимости. Если нет возможных исправлений или суммарная стоимость исправления слишком высока, то возвращается ошибка.

Преимуществом таких алгоритмов является то, что они могут быть применены для любых восходящих анализаторов и полностью не зависят от грамматики. К недостаткам можно отнести то, что для достижения наибольшего эффекта эти алгоритмы должны подстраиваться под нужную грамматику.

2.3. Локальная коррекция

Алгоритмы этого класса отличаются от алгоритмов предыдущего класса количеством рассматриваемых символов во входной строке. Алгоритм модифицирует входную цепочку так, чтобы был принят как минимум один символ. Встретив ошибку, анализатор вычисляет по своему состоянию некоторое множество допустимых в этом состоянии символов (*acceptable-set*). Затем пропускаются все символы входной цепочки, пока не встретится символ из этого множества. Затем состояние анализатора изменяется таким образом, чтобы символ, который не был пропущен, был допустимым. Существует несколько методов, различающихся способом вычисления множества допустимых символов и изменения состояния анализатора.

Преимуществом алгоритмов данного класса является их независимость от языка. В отличие от рассмотренных выше, они легко

могут быть «подстроены» под определенные ситуации. Анализаторы используют стандартные управляющие таблицы и незначительное количество дополнительной информации. Нет необходимости изменять грамматику языка.

Примером этого класса может служить алгоритм, описанный в статье [12].

2.4. Глобальная коррекция

В отличие от предыдущих классов, эти алгоритмы используют при анализе бóльшую часть контекста. Таким образом, выбираются наиболее правильные исправления. Но данные алгоритмы неэффективны, поскольку корректные и некорректные программы требуют одинаковых затрат. Как следствие, очень много времени уходит впустую — на обработку правильных цепочек.

Поэтому чаще рассматривают класс региональных изменений, что является компромиссом между локальной и глобальной обработкой ошибок, и таким образом принимается во внимание только фиксированная, ограниченная часть контекста для каждой ошибки.

В связи с ростом производительности компьютеров данный класс алгоритмов приобрёл наибольшую популярность. Примерами этого класса могут служить алгоритмы, описанные в статьях [7, 11, 13].

2.5. Выводы

С одной стороны, ни один из рассмотренных классов алгоритмов не удовлетворяет всем поставленным требованиям. Основным недостатком специальных алгоритмов является отсутствие автоматизации. Общим недостатком остальных алгоритмов является зависимость от грамматики: для улучшения результатов требуется настройка алгоритма по конкретной грамматике. Кроме того, приходится делать выбор между быстродействием (локальные коррекции) и качеством исправления ошибок (глобальные коррекции).

С другой стороны, наиболее подходящим методом оказался метод пустых ячеек. Единственный его недостаток — отсутствие автоматизации, т. е. невозможность автоматически выбрать исправляющее действие. Однако в поставленной задаче такой выбор существенно упрощается, поскольку требуется только добавление терминалов. Поэтому именно данный метод был взят за основу предлагаемого алгоритма исправления ошибок.

3. Понятия и термины

Прежде чем приступить к описанию предлагаемого алгоритма, опишем принципы восходящего анализа и устройства управляющих таблиц LR(k)- и LALR(k)-анализаторов.

Восходящий анализатор (bottom-up parsing) предназначен для построения дерева разбора, начиная с листьев и двигаясь вверх к корню дерева разбора. Мы можем представить себе этот процесс как «свертку» исходной строки w к начальному нетерминалу грамматики. Каждый шаг свертки заключается в сопоставлении некоторой подстроки w и правой части какого-то правила грамматики и замене этой подстроки на нетерминал, являющийся левой частью правила. Если на каждом шаге подстрока выбирается правильно, то в результате мы получим правый вывод строки w [3].

3.1. LR(k)- и LALR(k)-анализ

LR(k) означает, что:

- **L**: входная цепочка обрабатывается слева направо (left-to-right parse);
- **R**: выполняется правый вывод (rightmost derivation);
- **k**: не более k символов цепочки используются для принятия решения (k-token lookahead).

При LR(k)-анализе применяется метод *«перенос-свертка» (shift-reduce)*. Этот метод использует магазинный автомат. Символы входной цепочки переносятся в магазин до тех пор, пока на вершине магазина не накопится цепочка, совпадающая с правой частью какого-нибудь из правил (операция «перенос»). Далее все символы этой цепочки извлекаются из магазина и на их место помещается нетерминал, находящийся в левой части этого правила (операция «свертка»). Входная цепочка допускается автоматом, если после переноса последнего символа входной цепочки и выполнения операции «свертка» в магазине окажется только начальный нетерминал грамматики.

LALR(k)-анализ означает LP(0)-анализ с предварительным просмотром k терминалов (Look-Ahead of k tokens). Класс языков, распознаваемых LALR-анализаторами, несколько меньше LR(k)-языков, однако большинство стандартных синтаксических конструкций языков программирования легко выражается с помощью LALR-грамматики. Также таблицы LALR-анализатора значительно меньше канонических LR-таблиц, что делает анализатор менее

требовательным к памяти. Поэтому на практике именно LALR-анализаторы получили наибольшее распространение.

Для того чтобы избежать лишних проверок и возможных проблем, в случае, когда начальный нетерминал грамматики выведен, но разбор входной цепочки еще не закончен, обычно в конец входной цепочки добавляется маркер конца строки (end mark). В данной статье для этой цели мы будем использовать символ `$`. Однако в других статьях может использоваться другой символ, например «#» [10]. Также в грамматику будем добавлять новое правило: `S'` → `S $`, где `S` — начальный нетерминал исходной грамматики, а `S'` — начальный нетерминал новой грамматики с добавленным символом конца строки.

3.2. Управляющая таблица LALR(k)-анализатора

Управляющая таблица анализатора (табл. 1) состоит из двух частей: *таблицы действий (action table)* и *таблицы переходов (go to table)*. Строкам в управляющей таблице соответствуют состояния автомата, а столбцам — терминалы в таблице действий и нетерминалы в таблице переходов.

В качестве примера рассмотрим управляющую таблицу анализатора для следующей грамматики:

```
0  S → E $
1  E → E + E
2  E → a
```

Если автомат находится в состоянии i и во входной цепочке присутствует символ t, то мы находим в управляющей таблице соот-

Таблица 1. Пример управляющей таблицы анализатора

	action			go to	
	a	+	$	E	S
0	s1			2	
1		r2	r2		
2		s2	s3		
3	accept				
4	s1			5	
5		r1	r1		

Таблица 2. Пример разбора строки «a+a»

Стек	Входная цепочка	Действие
0	a+a$	shift ‘a’, go to 1
0 a 1	+a$	reduce rule 2 (go to 2)
0 E 2	+a$	shift ‘+’, go to 4
0 E 2 + 4	a$	shift ‘a’, go to 1
0 E 2 + 4 a 1	$	reduce rule 2 (go to 5)
0 E 2 + 4 E 5	$	reduce rule 1 (go to 2)
0 E 2	$	shift ‘$’, go to 4
0 E 2 $ 3		accept

ветствующее действие и выполняем его. Например, для состояния `0` и символа «`a`» это действия `shift 'a'`, `go to state 1`, т. е. перенос символа «`a`» на вершину стека и переход автомата в состояние `1`. Для состояния `1` и символа «`+`» это действие `reduce by rule 2`, т. е. свертка по правилу `2`. В этом случае со стека снимается n символов, где n — длина правой части правила `2`, а состояние автомата «откатывается» на n состояний назад. Далее совершается переход в соответствии с текущим состоянием. Например, при считывании строки, начинающейся с «`a+`», сначала будет сделан перенос символа «`a`» и переход в состояние `1`, потом выполнится свертка по правилу `2`, со стека будет снят терминал `a`, автомат перейдет в состояние `0`, на стек будет помещен нетерминал `E`. Далее в соответствии с таблицей переходов автомат перейдет в состояние `2`. При переходе в состояние `3` анализатор завершает работу, допустив входную строку (подробности см. табл. 2).

4. Описание базового алгоритма

Как уже было сказано выше, предлагаемый алгоритм будет рассмотрен на примере LALR-анализаторов. Однако нужно заметить, что он подойдет для любого анализатора типа «перенос-свертка», обладающего управляющей таблицей, аналогичной таблицам LALR-анализаторов, например для LR-, SLR-, GLR-анализаторов.

4.1. Синтаксически незначимые терминалы

Допустим, что в action-таблице встречается такая строчка:

состояние i	error	...	error	shift, go to j	error	...

То есть в состоянии i транслятор принимает только некоторый символ t и выполняет переход в состояние j. Будем называть такой терминал *синтаксически незначимым для данного состояния*, а грамматики, содержащие такие терминалы, — *избыточными с точки зрения построения LALR-трансляции*. При корректном вводе в этом состоянии мы можем встретить только символ t. Поэтому если транслятор находится в состоянии i, а во входной строке находится другой символ, то транслятор выдаст ошибку. Можно попытаться обработать ее, предположив, что данный терминальный символ был пропущен. Для этого определим новое действие `push t, go to j`: положить на стек символ t и перейти в состояние j. Такое действие будем называть *проталкиванием символа* t. Поместим его во все пустые ячейки состояния i.

Хотим подчеркнуть, что мы рассматриваем только action-таблицу. Транслятор обращается к goto-таблице только при свертках, для определения нового состояния. Пустые ячейки состояния в goto-таблице указывают не на ошибки ввода, а на то, что в этом состоянии на стеке не может быть соответствующего нетерминала.

Рассмотрим предлагаемый алгоритм на примере грамматики № 1, порождающей строку вида «`a(+a)*`»:

```
0  S → E $
1  E → E + E
2  E → a
```

Для наглядности на рисунке приведена диаграмма состояний соответствующего LALR-транслятора.

Таблица 3. Управляющая LALR-таблица для грамматики № 1

	a	+	$	E	S
0	s1			2	
1		r2	r2		
2		s2	s3		
3	accept				
4	s1			5	
5		r1	r1		

Как видно из табл. 3, синтаксически незначимым является терминал «`a`». Модифицированный с помощью предложенного алго-

Таблица 4. Модифицированная управляющая таблица для грамматики № 1

	a	+	$	E	S
0	s1	p'a'1	p'a'1	2	
1		r2	r2		
2		s2	s3		
3	accept				
4	s1	p'a'1	p'a'1	5	
5		r1	r1		

Таблица 5. Разбор строки «a+» анализатором, описанным в табл. 4

Стек	Входная цепочка	Действие
0	a+$	shift, go to 1
0 a 1	+$	reduce rule 2
0 E 2	+$	shift, go to 4
0 E 2 + 4	$	push 'a', go to 1
0 E 2 + 4 a 1	$	reduce rule 2
0 E 2 + 4 E 5	$	reduce rule 1
0 E 2	$	shift, go to 4
0 E 2 $ 3		accept

Таблица 6. Модифицированная управляющая таблица для грамматики № 1

	a	+	$	E	S
0	s1	p'a'1	p'a'1	2	
1		r2	r2		
2	p'+'2	s2	s3		
3	accept				
4	s1	p'a'1	p'a'1	5	
5		r1	r1		

Таблица 7. Разбор строки «aa» анализатором, описанным в табл. 6

Стек	Входная цепочка	Действие
0	aa$	shift, go to 1
0 a 1	a$	error

Состояния автомата LALR-компилятора грамматики № 1.

Таблица 8. Модифицированная управляющая таблица для грамматики № 1

	a	+	$	E	S
0	s1	p'a'1	p'a'1	2	
1	r2	r2	r2		
2	p'+'2	s2	s3		
3	accept				
4	s1	p'a'1	p'a'1	5	
5	r1	r1	r1		

ритма анализатор[4], принимающий грамматику № 1, представлен в табл. 4. Пример разбора строки с ошибкой представлен в табл. 5.

Но логичнее было бы предположить, что незначимым является «+». Этот терминальный символ принимается только в состоянии 2. Также в этом состоянии допускается символ конца строки. Поскольку символ конца строки является искусственным (изначально его не было в грамматике), постольку мы можем с уверенностью сказать, что он встретится только один раз и является последним символом строки. Поэтому если в состоянии 2 мы встретим символ a, то мы с уверенностью можем сказать, что пропущенным мог быть только символ «+». Поэтому символ «+» мы также будем называть синтаксически незначимым терминалом.

Однако в данном случае простого добавления «проталкивания»

[4]Полученный анализатор формально не является LALR-транслятором. В дальнейшем для краткости такие трансляторы будем называть модифицированными.

Таблица 9. Разбор строки «aa» анализатором грамматики, описанным в табл. 8

Стек	Входная цепочка	Действие
0	aa$	shift, go to 1
0 a 1	a$	reduce rule 2
0 E 2	a$	push '+', go to 4
0 E 2 + 4	a$	shift, go to 1
0 E 2 + 4 a 1	$	reduce rule 2
0 E 2 + 4 E 5	$	reduce rule 1
0 E 2	$	shift, go to 4
0 E 2 $ 3		accept

будет недостаточно. При разборе строки анализатором, представленным в табл. 6, ошибка возникнет в состоянии 1 (табл. 7). Ошибка возникает из-за того, что LALR-анализатор предполагал свертку только при обнаружении во входной строке символов «+» и `$`. Но хотелось бы, чтобы анализатор принимал строки с пропущенным символом «+». Для этого достаточно все пустые ячейки в состояниях, в которых определена свертка для символа «+», заменить на такие же свертки (табл. 8). В этом случае анализатор сможет обработать цепочки, в которых пропущен терминал «+» (табл. 9). Заметим, что многие реализации LALR-анализаторов выполняют свертку в состоянии в том случае, если для текущего символа не определено действие, и выбирают ее из всех возможных сверток в данном состоянии.

Обобщим эти выводы: если в некотором состоянии i анализаторов принимает только символ t (либо символ t и символ конца строки) и переходит в состояние j, то все пустые ячейки этого состояния могут быть заменены на `push t, go to j`. При этом все пустые ячейки в состояниях, в которых определена свертка для символа t, должны быть заменены на такие же свертки.

4.2. Семантически незначимые терминалы

При свертке LALR-анализатор вычисляет s-атрибуты [1] сворачиваемых терминалов, если они участвуют в вычислении s-атрибута полученного нетерминала.

Допустим, что атрибутная грамматика не содержит семантических правил, в вычислениях которых участвует s-атрибут данного терминала. Такие терминалы будем называть *семантически незначимыми*. В этом случае нам неважно, какой терминальный символ

будет помещен на стек (при свертке мы не смотрим на то, какие лексемы находятся на стеке, а просто убираем несколько символов). Поэтому на стек можно поместить любой терминал. Для этой цели в грамматику желательно добавить *пустой символ*. Тогда нам не понадобится генерация атрибута этого токена.

Если же терминальный символ является семантически значимым, то просто поместить на стек пустой символ нельзя. В этом случае нужно уметь порождать s-атрибут терминала с учетом контекста. Например, в объектно-ориентированных языках программирования это означает, что у объекта, представляющего данный терминал, должен быть предусмотрен некий конструктор от сокращенного контекста.

4.3. Обсуждение базового алгоритма

Приведём некоторые рассуждения и замечания, которые предназначены акцентировать внимание читателя на важнейших свойствах алгоритма.

- Данный алгоритм не порождает новых цепочек целевого языка. Некорректная строчка, принимаемая ослабленным транслятором, вызовет проталкивание одного или нескольких терминальных символов, что, по своей сути, является добавлением терминального элемента во входную цепочку и его перенос на стек. А следовательно, для любой некорректной цепочки, принимаемой ослабленным транслятором, можно представить корректную цепочку с таким же результатом трансляции.
- Стоит заметить, что предложенный алгоритм не модифицирует грамматику, не требует вмешательства программиста, не зависит от входной грамматики и не модифицирует входную строку.
- На первый взгляд может показаться, что данный алгоритм просто рассматривает незначимые терминалы как опциональные. Например: к правилу $\mathtt{X} \to \alpha\mathtt{a}\beta$ добавляется правило $\mathtt{X} \to \alpha\beta$. На деле это не так. Во-первых, в предлагаемом алгоритме изменяется не правило грамматики, а поведение автомата в некотором состоянии, что является определенным преимуществом (например, не нужно перестраивать управляющую таблицу). Во-вторых, добавление новых правил может привести к неоднозначностям в грамматике, в то время как

предлагаемый алгоритм к неоднозначностям привести не может, поскольку изменяются лишь пустые ячейки управляющей таблицы.

Например, рассмотрим грамматику № 2 (детали см. табл. 10):

```
0 S → A $
1 A → B C
2 A → B C C
3 B → y y
4 C → y
```

Таблица 10. Управляющая LALR-таблица для грамматики № 2

	y	$	A	B	C	S
0	s1		2	3		
1	s4					
2		s5				
3	s6				7	
4	r3					
5	accept					
6	r4	r4				
7	s6	r2	8			
8		r1				

Таблица 11. Модифицированная управляющая таблица для грамматики № 2

	'y'	$	A	B	C	S
0	s1	p'y'1	2	3		
1	s4	p'y'4				
2		s5				
3	s6	p'y'6			7	
4	r3	r3				
5	accept					
6	r4	r4				
7	s6	r2	8			
8	r1	r1				

Из табл. 11 видно, что синтаксически незначимым является терминал «y» в состояниях 0 и 1, что соответствует символам «y» в правиле 3. Но добавление правила B → y приведет к конфликту переноса-свертки в состоянии 1. В то же время строка «yy» легко разбирается модифицированным транслятором (табл. 12).

Таблица 12. Разбор строки «yy» анализатором, описанным в табл. 11

Стек	Входная цепочка	Действие
0	yy$	shift, go to 1
0 y 1	y$	shift, go to 4
0 y 1 y 4	$	reduce rule 3
0 B 3	$	push 'y', go to 6
0 B 3 y 6	$	reduce rule 4
0 B 3 C 7	$	reduce rule 2
0 A 2	$	shift, go to 5
0 A 2 $		accept

Также незначимым является терминал «y» в состоянии 3, соответствующий символу «y» из правила 4. Но переносу символа «y» из правила 4 еще соответствует состояние 6, в котором данный терминал не является синтаксически незначимым. Из этого следует, что добавление нового правила неравносильно предлагаемому алгоритму даже в том случае, если оно не внесет конфликтов в грамматику. Предложенный алгоритм модифицирует не всё правило, а лишь часть соответствующих ему состояний.

5. Обобщение и совершенствование базового алгоритма

Базовый алгоритм достаточно прост для понимания, но оказался весьма непрактичным с точки зрения реализации. В данном разделе предлагаются соответствующие усовершенствования.

5.1. Действия по умолчанию

Как правило, состояния представлены списком действий (для каждого символа — свое действие). Это позволяет не хранить в памяти пустые ячейки. Поэтому добавление новых действий в некотором состоянии для всех терминалов, в которых встречается ошибка,– не очень удачная идея, поскольку размер списков действий у таких состояний увеличится с единицы до количества терминалов в грамматике. Гораздо удобнее и экономичнее с точки зрения используемой памяти определить действие по умолчанию, которое будет вызываться в том случае, если не было найдено обычное действие для терминального символа. Также из состояния можно удалить все действия, совпадающие с действием по умолчанию для данного состояния (см. табл. 13).

Таблица 13. Модифицированная управляющая таблица для грамматики № 1

	a	+	$	E	S	default
0	s1			2		push 'a', go to 1
1						reduce rule 2
2		s2	s3			push '+', go to 2
3						accept
4	s1			5		push 'a', go to 1
5						reduce rule 1

5.2. Состояния с несколькими переносами и свертками

Алгоритм, предложенный выше, модифицирует только те состояния, в которых определено ровно одно действие — перенос. А что можно сделать в других случаях?

Пусть в некотором состоянии есть n переносов (для символов $t_1, \ldots, t_n, n \geq 2$), но нет ни одной свертки. В таком состоянии для каждой пустой ячейки также можно определить действие `push, go to`. Но в этом случае нужно выбрать символ из $[t_1, \ldots, t_n]$, который будем помещать на стек. Сделать это можно тремя способами:

1) предложить пользователю выбрать терминал для каждой ячейки;
2) предложить пользователю определить множество Closers, которое содержит все закрывающие символы (например, для языка Pascal это символы «)», «;», *end*), и автоматически выбирать терминал из этого множества;
3) попытаться проанализировать, какой терминал был пропущен на самом деле.

Теперь рассмотрим состояния, в которых есть n переносов (для символов $t_1, \ldots, t_n$, $n \geq 1$) и m сверток, где $m \geq 1$. В данном случае также можно выбрать некоторый терминал из $[t_1, \ldots, t_n]$, чтобы определить действие `push, go to`. Но такое действие вносит изменения во входную цепочку. В то же время мы точно знаем, что в указанном состоянии можно выполнить свертку. Поэтому пустые ячейки этого состояния логичнее было бы заменить на одну из возможных сверток. Способы выбора свертки аналогичны способам выбора действия `push, go to`. Однако можно добавить более простой способ — автоматический выбор любой свертки, например, первой или наиболее часто встречающейся в этом состоянии.

Если же в состоянии есть только свертки, но нет переносов, то мы можем выбрать любую свертку.

Теперь обобщим все рассуждения. Пусть есть некоторое состояние, содержащее n переносов (для символов $t_1, \ldots, t_n$, $n \geq 1$) и m сверток по правилам $r_1, \ldots, r_k$, где $n + m \geq 1$, $k \leq m$ (поскольку в одном состоянии может быть несколько сверток по одному правилу). Пусть $m \geq 1$. Тогда все пустые ячейки action-таблицы можно заменить на свертку по правилу r_i, где $i \in [1, \ldots, k]$. Если $m = 0$, то для всех пустых ячеек можно определить действие `push` $\mathtt{t}_i$`, go to`, где $i \in [1, \ldots, n]$.

Покажем, что в случае множественных переносов нельзя выбирать любой из них. Автоматический выбор случайного терминала в этом случае может привести к бесконечной рекурсии. Рассмотрим, например, грамматику № 3:

```
0 S → T $
1 T → ( E )
2 E → E , E
3 E → a
```

В табл. 14 представлен модифицированный транслятор, принимающий грамматику № 3. В состоянии `4` был выбран первый перенос – перенос запятой. Теперь рассмотрим разбор строки «`(a;`» (табл. 15). После считывания символа «`a`» из входной строки мы проталкиваем на стек терминал «`,`», потом терминал «`a`», выполняем свертки по правилам `3` и `2` и повторяем эту последовательность операций. В результате получаем бесконечную рекурсию.

Примечание. В том случае, если все проталкиваемые терминальные символы семантически незначимы, либо мы умеем порождать для них s-атрибут, замена всех пустых ячеек приведет к тому, что транслятор будет принимать любые строчки.

5.3. Анализ пропущенных терминалов

Теперь рассмотрим способ, позволяющий определить пропущенный терминальный символ.

Рассмотрим грамматику № 4, принимающую строчки «`ac`» и «`bd`»:

```
0 S → E $
1 E → a c
2 E → b d
```

Таблица 14. Модифицированная управляющая таблица для грамматики № 3

	a	,	(	)	;	$	T	E	S	default
0				s1			2			push '(', go to 1
1	s3							4		push 'a', go to 3
2						s5				push '$', go to 5
3										reduce rule 3
4		s6	s7							push ',', go to 6
5										accept
6	s3							8		push 'a', go to 3
7					s9					push ';', go to 9
8										reduce rule 2
9										reduce rule 1

Таблица 15. Разбор строки «(a;» анализатором, описанным в табл. 15

Стек	Входная цепочка	Действие
0	(a;$	shift, go to 1
0 (1	a;$	shift, go to 3
0 (1 a 4	;$	reduce rule 3
0 (1 E 4	;$	push ',', go to 6
0 (1 E 4 , 6	;$	push 'a', go to 3
0 (1 E 4 , 6 a 3	;$	reduce rule 3
0 (1 E 4 , 6 E 8	;$	reduce rule 2
0 (1 E 4	;$	push ',', go to 6
0 (1 E 4 , 6	;$	push 'a', go to 3
0 (1 E 4 , 6 a 3	;$	reduce rule 3
0 (1 E 4 , 6 E 8	;$	reduce rule 2
...	...	...

Как видно из табл. 16, анализатор примет также строчки «a» и «b». Можно расширить принимаемый язык словом «c» или «d», добавив в состояние 0 проталкивание по умолчанию символа «a» и переход в состояние 1 или проталкивание по умолчанию символа «b» и переход в состояние 2 соответственно. Но хотелось бы добавить возможность разобрать оба слова.

Рассмотрим перенос символа «a» в состоянии 0. Он переводит анализатор в состояние 1, в котором принимается только символ «c». При этом анализатор не предусматривает никакого действия для терминала «c» в состоянии 0 (за исключением действия по умолчанию). Поэтому в состоянии 0 можно определить действие для символа «c»: поместить на стек символ «a» и перейти в состояние 1. Аналогично для символа «d» — поместить на стек символ «b» и перейти в состоя-

Таблица 16. Модифицированная управляющая таблица для грамматики № 4

	a	b	c	d	$	E	S	default
0	s1	s2				3		
1			s4					push 'c', go to 4
2				s5				push 'd', go to 5
3					s6			push '$', go to 6
4								reduce rule 1
5								reduce rule 2
6								accept

ние 2. В качестве действия по умолчанию можно выбрать любое проталкивание — в результате мы сможем разобрать и пустую строку.

Обобщим этот результат, используя нотацию, принятую в учебнике Б. К. Мартыненко [2]. Пусть существуют состояния i, j и терминалы t_1, t_2 такие, что в состоянии i нет сверток, $f_i(t_1) =$ перенос j, $f_i(t_2) = error$, $f_j(t2) \neq error$ или определено действие по умолчанию. Тогда в состоянии i можно определить действие для терминала t_2 – проталкивание символа t_1 и переход в состояние j.

6. Примеры использования алгоритма

Для реализации предложенного алгоритма был выбран один из самых известных генераторов парсеров с открытым исходным кодом — *Bison* (версия 2.3-1)[5].

Для грамматики, приведенной выше, анализатор выдал следующий результат:

```
state 0:
accept only 'a' symbol; default action is push 'a', go to 1
state 2:
accept only '+' symbol; default action is push '+', go to 4
state 4:
accept only 'a' symbol; default action is push 'a', go to 1
total 3 default pushes
```

[5]GNU Bison — приложение, предназначенное для автматического создания синтаксических анализаторов по данному на вход описанию грамматики. Данное приложение является OpenSource ПО и разрабатывается в рамках проекта GNU. Более подробную информацию можно найти на официальном сайте проекта: http://www.gnu.org/software/bison/

6.1. Анализ грамматики ANSI C

В качестве примера была выбрана грамматика языка ANSI C[6]. Оказалось, что 29 состояний из 349 содержат синтаксически незначимые терминалы.

Рассмотрим наиболее интересные результаты, которые демонстрируют, как на этапе разработки языка можно контролировать его синтаксическую чистоту и краткость.

В основном в ANSI C лишними являются различные скобки. Например, в состоянии 89:

```
state 89
 192 selection_stmt: IF . '(' expression ')' stmt
 193    | IF . '(' expression ')' stmt ELSE stmt
   '('  shift, and go to state 175
   $default  push symbol '(', and go to state 175
```

В этом состоянии предполагается, что на входе будет открывающая скобка, которая избыточна. А закрывающую скобку можно рассматривать как аналог терминала `THEN` в языке Pascal, который разделяет условное выражение и последовательность операторов.

Также встречаются лишние «`;`». Например, терминал «`;`» не нужен после слов `CONTINUE` и `BREAK`.

Рассмотрим цикл `DO...WHILE`. Соответствующее правило выглядит так:

```
iteration_stmt DO stmt WHILE '(' expression ')' ';'
```

Судя по результатам анализа, лишними (синтаксически незначимыми) являются терминалы `WHILE`, «`(`», «`;`».

Действительно, после слова `DO` мы считываем тело цикла, потом обязательно идет — «`WHILE (`», и после — символ «`;`». Символ «`)`» действительно не является незначимым из-за правила

```
expression expression ',' assignment_expression.
```

[6]ANSI C является стандартом языка C, разработанным Американским национальным институтом стандартов (American National Standards Institute, ANSI). Этот стандарт широко используется на практике для создания переносимого на разные платформы программного кода. Описание грамматики ANSI C в формате YACC можно найти на сайте университета Linkoping (http://www.lysator.liu.se/c/ANSI-C-grammar-y.html)

То есть в соответствующем состоянии ожидается как закрывающая скобка, так и запятая.

Таким образом, цикл может быть представлен строчкой: «`DO stmt expression ')'`». На первый взгляд наличие только закрывающей скобки выглядит не очень красиво, но на её месте вполне может быть и «`;`». Справедливо заметить, что `WHILE` визуально отделяет условие от тела цикла и удобен программисту, хоть и бесполезен для транслятора, но скобки являются абсолютно лишними. Поэтому цикл вполне мог бы выглядеть как «`DO stmt WHILE expression ';'`», что очень напоминает «`REPEAT .. UNTIL`» в Pascal.

Также необходимо обратить внимание на сообщения анализатора о терминале `IDENTIFIER`. Например:

```
state 94
 199 jump_stmt: GOTO . IDENTIFIER ';'
   IDENTIFIER  shift, and go to state 180
   $default  push symbol IDENTIFIER, and go to state 180
```

Такое сообщение говорит не о том, что терминал может быть опущен во входной строке, а о том, что в корректной входной строке только он и может быть считан в этом состоянии. Это пример синтаксически незначимого, но семантически значимого терминала. Дело в том, что на данный момент предложенный алгоритм реализован не полностью.

Также была проанализирована грамматика ANSI-ISO Pascal[7]. В ней 59 состояний из 409 содержат синтаксически незначимые терминалы.

Заключение

Предложенный алгоритм анализа языков, порождаемых LALR-грамматиками, позволяет определить предложения и терминалы, которые могли бы быть опущены без изменения семантического значения языка. Полученная таким образом информация используется в алгоритме построения LALR-транслятора с функцией ис-

[7]ANSI-ISO Pascal является стандартом языка Pascal, принятым международной организацией по стандартизации (International Organization for Standardization, ISO). Описание грамматики ANSI-ISO Pascal в формате YACC можно найти на сайте http://www.moorecad.com/standardpascal/pascal.y

правления ошибок рассматриваемого класса. Результаты анализа также могут указать на наличие определенного рода избыточности в рассматриваемом языке. Таким образом, предлагаемый алгоритм можно использовать при разработке новых грамматик. Это позволит создать язык, в котором все правила очищены от использования терминалов, не влияющих на трансляцию, выявив их еще на этапе проектирования грамматики. Но, безусловно, объявление того или иного вхождения терминала избыточным не является рекомендацией по его удалению, поскольку такая избыточность может улучшать восприятие текстов на данном языке программирования для его пользователей.

В настоящей статье осталась незатронутой проблема автоматизации обработки семантически незначимых элементов и мы видим продолжение развития темы в детальном исследовании означенной проблемы с дальнейшим обобщением от токенов к семантически незначимым цепочкам.

Список литературы

[1] *Кнут Д., Хоор Ч., Льюис П.* Семантика языков программирования. М., 1980.

[2] *Мартыненко Б. К.* Языки и трянсляции. СПб., 2004.

[3] *Вояковская Н. Н., Москаль А. Е., Булычев Д. Ю., Терехов А. А.* Разработка компиляторов на платформе .NET: Курс лекций. СПб., 2001.

[4] *Терехов А. Н., Эрлих Л. А., Терехов А. А.* История и архитектура проекта Rescueware // Автоматизированный реинжиниринг программ. СПб., 2000. С. 7–19.

[5] *Чемоданов И. С., Дубчук Н. П.* Генератор синтаксических анализаторов для решения задач автоматизированного реинжиниринга программ // Системное программирование. 2006. С. 268–296.

[6] *Aho A., Sethi R., Ullman J.* Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.

[7] *Burke M. G., Fisher G. A.* A Practical Method for *LR* and *LL* Syntactic Error Diagnosis and Recovery // ACM Transactions on Programming Languages and Systems. April 1987. Vol. 9. P. 164–197.

[8] *Conway R. W., Wilcox T. R.* Design and Implementation of a Diagnostic Compiler for PL/I // Communications of the Association for Computing Machinery. March 1973. Vol. 16. P. 169–179.

[9] *Degano P., Priami C.* Comparison of Syntactic Error Handling in LR Parsers // Software-Practice and Experience. June 1995. Vol. 25. P. 657–679.

[10] *Grune D., Jacobs C.* Parsing Techniques — A Practical Guide. Ellis Horwood Limited, 1990.

[11] *Mckenzie B. J., Yeatman C., De Vere L.* A Practical Method for *LR* and *LL* Syntactic Error Diagnosis and Recovery // ACM Transactions on Programming Languages and Systems. July 1995. Vol. 17. P. 672–689.

[12] *Rohrich J.* Methods for the Automatic Construction of Error Correcting Parsers // Acta Informatica. February 1980. Vol. 13. P. 115–139.

[13] *Vilares M., Darriba V. M., Ribadas F. J.* Regional Least-Cost Error Repair // Implementation and Application of Automata. LNCS. Vol. 2088. 2001. P. 293–301.