Санкт-Петербургский Государственный
Университет
Математико-механический факультет

Кафедра системного программирования

Построение ослабленного LALR-транслятора на основе анализа грамматики на избыточность

Дипломная работа студента 545 группы
Ефимова Андрея Александровича

Научный руководитель	..... /подпись/	ст. преп. Я.А. Кириленко
Рецензент	..... /подпись/	к.ф.-м.н. А.С. Лукичев
“Допустить к защите” заведующий кафедрой	..... /подпись/	д.ф.-м.н., проф. А.Н. Терехов

Санкт-Петербург
2008

Содержание

1 Введение

Данная работа посвящена автоматическому построению ослабленного транслятора для предупреждения ошибок трансляции, вызванных неточностью спецификации языка.

При разработке системы автоматизированного реинжиниринга [3] программных комплексов Relativity Modernization Workbench [13] возникла проблема, заключающаяся в неточности спецификации анализируемых языков. Документация на языки программирования, выпущенная вместе с компиляторами, со временем устаревает, не всегда своевременно обновляется.

Вследствие некоторой "избыточности" грамматики входного языка с точки зрения построения трансляции, может возникнуть отличие в поведении оригинальных компиляторов от спецификации синтаксиса входного языка. Встречаются такие ситуации, когда в документации ключевое слово объявлено как обязательное, но не влияет на трансляцию. В то же время оригинальные компиляторы, выпущенные вместе с такой документацией, позволяют опускать такие ключевые слова.

При автоматизированном реинжиниринге такое различие создает ряд проблем при поддержке старых исходных текстов. А именно, трансляторы, написанные по спецификации языка, не принимают исходные тексты программ, успешно транслируемые оригинальными компиляторами. Такие ошибки невозможно выявить с помощью тестирования, основанного на спецификации языка.

Поэтому было бы полезно по грамматике, заданной спецификацией, анализировать язык на избыточность с точки зрения построения транслятора. В результате мы сможем получить транслятор, который принимает некоторое расширение языка, позволяя опускать терминалы, не влияющие на результат трансляции.

Такой анализ можно будет применить и при разработке новых языков. Он позволит создать язык, в котором все правила очищены от использования терминалов, не влияющих на трансляцию, выявив их еще на этапе проектирования грамматики.

Целью данной работы является формализация задачи поиска незначимых терминалов и исследование способов автоматизации данного решения.

2 Постановка задачи

Искомое решение описанной в предыдущей главе проблемы должно основываться на предположении о возможной избыточности языка. Формальное определение избыточности грамматики с точки зрения построения трансляции дано в главе 4. Пока можно сказать, что предлагаемое решение позволит в определенных случаях опускать во входной цепочке некоторые лексемы, добавляя их в процессе анализа.

Стоит отметить, что искомое решение должно 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. Транслятор, обладающий этими свойствами, будем называть ослабленной модификацией оригинального транслятора, или ослабленным транслятором. Трансляцию, реализуемую ослабленным транслятором, будем называть ослабленной. Кроме того, желательно, чтобы построение ослабленного транслятора выполнялось автоматически.

По своей сути построение ослабленного транслятора сводится к задаче обработки некорректных цепочек. Такие задачи решают с помощью алгоритмов реакции на ошибку. Различают 3 основных способа:

1. Обнаружение ошибок (*error detection*)
2. Восстановление после ошибок (*error recovery*)
3. Исправление ошибок (*error correction*)

Наименьшее, что можно потребовать от транслятора, это показать, что во входной строке есть синтаксическая ошибка (т.е. что строка не принадлежит языку, описанному данной грамматикой). Это называется *обнаружением ошибок*.

Для некоторых приложений этого может быть достаточно. Но, как правило, необходимо узнать о 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. Также вывод может включать некоторое

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

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, который был бы невозможен при синтаксически корректном входе. Это может привести к появлению наведенных ошибок.

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

Более подходящим вариантом является использование различных методов *исправления ошибок* [12, 11]. При исправлении ошибок анализатор преобразует входную строку в синтаксически корректную, удаляя, добавляя или изменяя терминальные символы. Стоит заметить, что такое преобразование не всегда приводит входную строчку к той, которая подразумевалась пользователем. Поэтому к исправлению, как правило, допускаются только простые ошибки [9]: например, отсутствие точки с запятой.

В задачах реинжиниринга желательно уметь получать трансляцию даже при синтаксически некорректном входе. Например, одним из механизмов, используемым при оценке сложности предстоящего проекта, является вычисление формальных метрик (инвентаризация) [15]. В этом случае достаточно дать хотя бы приблизительный результат для некорректной входной цепочки.

Поэтому и для поставленной цели наиболее удачным будет использование алгоритма исправления ошибок. В следующей главе будет рассмотрена классификация таких методов, а также некоторые наиболее известные алгоритмы.

Большинство современных генераторов парсеров реализуют LR, $LALR$, GLR алгоритмы разбора [4]. В данной работе решение поставленной задачи будет рассматриваться на примере $LALR$-трансляторов [5, 12]. Существует большое количество LALR-грамматик языков программирования в открытом доступе, что существенно упростило анализ актуальности предлагаемого решения.

Структура дипломной работы. В главе 3 дан обзор имеющихся алгоритмов исправления ошибок. В главе 4 описывается предлагаемый алгоритм исправления ошибок на основе избыточности грамматики. В главе 5 рассматриваются некоторые возможные улучшения

предложенного алгоритма. В главе 6 рассматривается реализация инструмента анализа избыточности грамматики на основе генератора парсеров Bison. В приложении A приведены результаты анализа грамматики ANSI C [6], в приложении B – для грамматики ANSI-ISO Pascal [7]. Результаты работы сформулированы в части 7.

3 Обзор методов исправления ошибок

Сформулируем еще раз, 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;

4) алгоритм должен автоматически строить ослабленный транслятор;

Рассмотрим основные классы методов исправления ошибок.

3.1 Специальные методы

Одной из главных характеристик алгоритмов исправления ошибок является независимость от языка. Однако, большинство старых алгоритмов приспособлялись для определенной грамматики. Специальные методы получили такое название, потому что не могут быть автоматически сгенерированы по грамматике. Основным недостатком таких методов является отсутствие автоматизации.

3.1.1 Метод пустых ячеек

Таблица синтаксического анализа содержит пустые ячейки, при обращении к которым обнаруживается ошибка. В этом методе 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. В нем отсутствует автоматическая поддержка со стороны инструмента, поскольку значительная часть работы возлагается на человека. В общем случае для каждой пустой ячейки программисту необходимо 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.

Одна из реализаций данного алгоритма рассматривается в работе [10].

3.1.2 Метод добавления продукций ошибок

Для того чтобы снизить затраты на разработку анализатора и избавиться от некоторых недостатков предыдущего алгоритма, был предложен метод добавления продукций ошибок. Этод метод основывается на расширении контекстно-свободной грамматики с помощью специальных продукций, называемых продукциями ошибок. Продукции ошибок – 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 достаточно общий, независимость от грамматики в нем не была достигнута.

3.1.3 Метод добавления терминалов ошибки

Терминал ошибки – специальный терминальный символ, добавляемый перед точкой 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.

3.2 Коррекция на уровне фраз

Алгоритмы этого класса, как правило, 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 способ 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, такой анализ может быть рекурсивным.

На втором этапе анализатор модифицирует входную цепочку или стек, используя информацию, собранную на первом этапе. Как правило, различным модификациям 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.

3.3 Локальная коррекция

Отличается от предыдущего класса количеством рассматриваемых символов во входной строке. Алгоритм модифицирует входную цепочку так, чтобы был принят как минимум один символ. Когда анализатор встречает ошибку, он 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 независимость от языка. В отличие от предыдущего класса они легко могут быть подстроены под определенные ситуации. Анализаторы используют стандартные управляющие таблицы плюс незначительное количество дополнительной информации. Нет необходимости изменять грамматику языка.

3.3.1 Режим паники

Самый простой метод, который хоть насколько-то эффективен. Множество допустимых символов определяется программистом и не изменяется в течение всего анализа строки. Как правило, состоит из символов, завершающих конструкции языка (например, стэйтмента для языков программирования). Для языка Pascal – ; и *end*. При возникновении ошибки ищется первый элемент из этого множества, затем анализатор должен перейти в состояние, в котором принимается этот символ. В случае LR-грамматики, из стека постепенно удаляются состояния, пока на вершине стека не окажется нужное.

Часто является частью более сложных алгоритмов. Например, см. пункт 3.4.1.

3.3.2 Допустимые множества, выведенные из продолжений

Допустим, что анализатор обладает свойством проверки корректности префикса. Пусть он обнаружил ошибку после прочтения префикса u. Из-за свойства проверки корректности префикса мы знаем, что u – начало некоторой строки, которая принимается данной грамматикой. Поэтому должно существовать некоторое продолжение w такое, что uw принимается анализатором. Найдем это w. Для всех его префиксов w' вычислим множество терминальных символов, которые будут приняты анализатором после того, как он примет w' и возьмем объединение этих множеств в качестве множества допустимых символов. Если некоторый символ a принадлежит этому множеству, то $uw'a$ – префикс некоторого предложения в языке. Заметим, что множество содержит все символы суффикса w, в том числе и маркер конца строки (поэтому анализатор может пропустить все символы неразобранной строки вплоть до маркера). Добавим во входную строку самый короткий префикс w, после которого мы сможем принять некоторый символ входной строки. Выведем сообщение о том, какие символы были пропущены и какие были добавлены. Возобновляем работу анализатора в месте ошибки, начиная с добавленных символов.

Более подробно рассматривается в работе [14].

3.4 Глобальная коррекция

В отличие от предыдущих классов эти алгоритмы используют большую часть контекста. Таким образом, выбираются наиболее правильные исправления. Такие алгоритмы неэффективны из-за того, что на корректные и некорректные программы требуются одинаковые затраты. Таким образом, очень много времени тратится зря на обработку правильных цепочек.

Поэтому чаще рассматривают подкласс региональных изменений, являющийся компромиссом между локальными и глобальными обработками ошибок. Они рассматривают только фиксированную, ограниченную часть контекста для каждой ошибки.

3.4.1 Метод синтаксической диагностики для LR- и LL-анализаторов

Сначала пытаемся применить простое исправление (*simple repair*) – когда можно удалить/добавить/изменить один терминал. Основная проблема – сложно найти терминал, изменение которого приведет к

синтаксически корректному входу. В худшем случае его уже может не быть на стеке, поскольку он был свернут в какой-то нетерминал. Поэтому может потребоваться *unparsing*, что занимает очень много времени). Если не получилось, то удаляем и/или добавляем несколько терминалов, чтобы закрыть одну или несколько конструкций – например, процедуру, цикл или стэйтмент. Для этого нужно определить множество терминалов $Closers$, которое содержит все закрывающие символы (например,), ;, *end*). Это называется *восстановлением контекста* (*scope recovery*).

Более подробно рассматривается в работе [9].

3.5 Заключение

Ни один из рассмотренных выше методов не удовлетворяет всем поставленным требованиям. Основным недостатком всех специальных методов является отсутствие автоматизации. Для остальных классов общим недостатком 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).

Наиболее подходящим методом, на мой взгляд, является метод пустых ячеек. Единственным его недостатком является отсутствие автоматизации, то есть невозможность автоматически выбрать исправляющее действие. Однако для поставленной задачи такой выбор существенно упрощается, поскольку требуется только добавление терминалов. Поэтому именно этот метод был взят за основу предлагаемого алгоритма исправления ошибок.

4 Описание алгоритма

Как уже было сказано ранее, предлагаемый алгоритм будет рассматриваться на примере $LALR$-трансляторов. Однако, стоит заметить, что он подойдет для любого транслятора типа перенос-свертка, обладающего управляющей таблицей аналогичной таблицам $LALR$-трансляторов. Например, для $LR(k)$, SLR, GLR.

4.1 Синтаксически незначимые терминалы

Допустим, что в *action*-таблице встречается такая строчка:

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

Т.е. 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.

Стоит заметить, 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`

Как видно из таблицы 1, синтаксически незначимым является терминал a. Модифицированный с помощью предложенного алгоритма

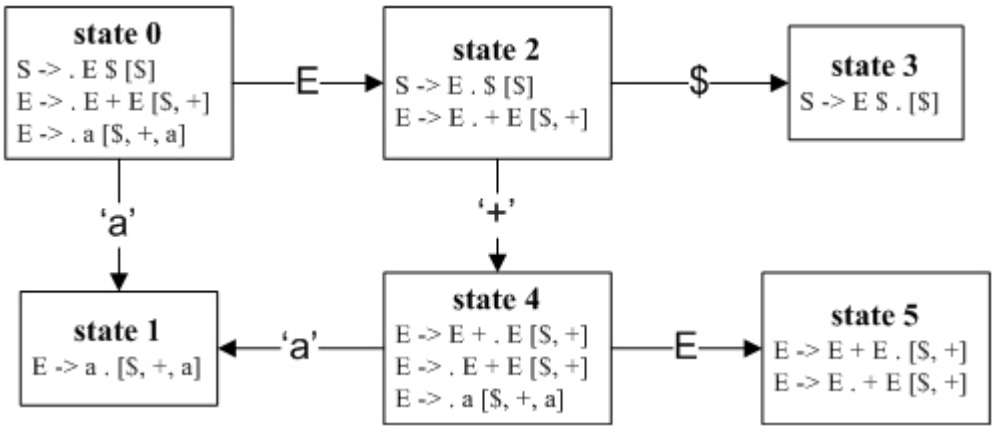

Рис. 1: Состояния автомата LALR-компилятора грамматики №1

Таблица 1: 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		

транслятор[2], принимающий грамматику №1, представлен в таблице 2. Пример разбора строки с ошибкой представлен в таблице 3

Но логичнее было бы предположить, что незначимым является +. Этот терминальный символ принимается только в состоянии 2. Также в этом состоянии может встретиться символ конца строки. Поскольку символ конца строки является искусственным (изначально его не было в грамматике), то мы можем с уверенностью сказать, что он может встретиться 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 будет недостаточно. При разборе строки транслятором, представленным в

[2]Полученный транслятор формально не является *LALR*-транслятором. В дальнейшем для краткости такие трансляторы будем называть модифицированными.

Таблица 2: 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 3: Разбор строки "a+" компилятором, описанным в таблице 2

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 4: 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 5: Разбор строки 'aa' компилятором грамматики, описанным в таблице 4

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

Таблица 6: 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		

Таблица 7: Разбор строки 'aa' компилятором грамматики, описанным в таблице 6

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

таблице 4, ошибка возникнет в состоянии 1: таблица 5. Она возникает из-за того, что $LALR$-транслятор предполагал свертку только при нахождении во входной строке символов + и \$. Но хотелось бы, чтобы транслятор принимал строки с пропущенным символом +. Для этого достаточно все пустые ячейки в состояниях, в которых определена свертка для символа +, заменить на такие же свертки. В этом случае транслятор сможет обработать цепочки, в которых пропущен терминал +: таблица 7. Заметим, что многие реализации $LALR$-трансляторов выполняют свертку в состоянии в том случае, если для текущего символа не определено действие, выбирая ее из всех возможных сверток в данном состоянии.

Обобщим эти выводы: если в некотором состоянии i транслятор 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 $LALR$-транслятор вычисляет s-атрибуты 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 [1] данного терминала. Такие терминалы будем называть *семантически незначимыми*. В этом случае нам не важно, 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 можно добавить *пустой символ*. В этом случае 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 Доказательство корректности алгоритма

Теперь покажем, что приведенный выше алгоритм порождает автомат, который принимает язык (M), являющийся надмножеством языка, описанного входной грамматикой (L). Для этого покажем, что любая строка языка L содержится в языке M. Пусть некоторая строка s содержится в языке L. Тогда при ее разборе транслятор языка L не обратится ни к одной пустой ячейке. Но транслятор языка M был получен из транслятора языка L лишь с помощью замен части пустых ячеек (ошибок) на проталкивания. Значит, и транслятор языка L при анализе строки s не будет обращаться ни к пустым ячейкам, ни к модифицированным. Поэтому он примет строку s, выполнив точно такой же разбор, и построит такую же трансляцию.

Кроме того, данный алгоритм не порождает новых цепочек целевого языка. То есть совпадают образы L и M относительно ослабленной трансляции. 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 значит, для любой некорректной цепочки, принимаемой ослабленным транслятором, можно представить корректную цепочку с такой же трансляцией.

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 $X \to \alpha a \beta$ добавляется правило $X \to \alpha\beta$. На самом деле, это не так. Во-первых, в предлагаемом алгоритме изменяется не правило грамматики, а поведение автомата в некотором состоянии, что является определенным преимуществом (например, ненужно 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:

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

Таблица 8: 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 9: 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 таблицы 9 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: таблица 10.

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 соответсвующих ему состояний.

Таблица 10: Разбор строки “yy’ транслятором, описанным в таблице 9

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

5 Возможные улучшения алгоритма

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, является не очень удачной идеей, поскольку размер списков действий у таких состояний увеличится с 1 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. Так же из состояния можно удалить все действия, совпадающие с действием по умолчанию для данного состояния. Например, управляющая таблица модифицированного транслятора, принимающего грамматику №1, будет выглядеть так: таблица 11

Таблица 11: 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

Алогритм, предложенный в главе 4, 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, ..., 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 $[t_1..t_n]$, который будем помещать на стек. Сделать это можно 3 способами:

1. Предложить пользователю выбрать терминал для каждой ячейки.

2. Предложить пользователю определить множество Closers 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, ..., t_n, $n \geq 1$) и m сверток, $n \geq 1$, $m \geq 1$. В этом случае также можно выбрать некоторый терминал из $[t_1..t_n]$, чтобы определить действие *push*, *go to*. Но такое действие вносит изменения во входную цепочку. В то же время мы точно знаем, что в данном 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*. Разве что 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, содержащие n переносов (для символов t_1, ..., t_n, $n \geq 1$) и m сверток по правилам r_1, ..., r_k, $n+m \geq 1$, $k \leq m$ (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 $i \in [1..k]$. Если $m = 0$, то для всех пустых ячеек можно определить действие *push* t_i, *go to*, где $i \in [1..n]$.

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

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

В таблице 12 представлен модифицированный транслятор, принимающий грамматику №3. В состоянии 4 был выбран первый перенос – перенос запятой. Теперь рассмотрим разбор строки “(a;”: таблица 13. После считывания символа a из входной строки мы проталкиваем на стек терминал ,, потом терминал a, выполняем

Таблица 12: 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 13: Разбор строки "(a;" анализатором, описанным в таблице 12

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
...	...	...

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 либо семантически незначимы, либо мы умеем порождать для них s-атрибут, то 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`

Как видно из таблицы 14, 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 состояние 2. В качестве действия

Таблица 14: 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

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 $\exists$ 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 Реализация анализатора грамматики на избыточность

Для исследования актуальности предложенного алгоритма был выбран один из самых известных генераторов парсеров с открытым исходным кодом – *Bison* (версия 2.3-1) [8]

В *Bison*'е $LALR$ таблицы устроены следующим образом. Автомат представлен списком состояний. У каждого состояния есть список действий. Действие представлено парой символ-число. Если на входе встречается данный символ, то выполняется связанное с ним действие n. В том случае, если число n положительное, выполняется перенос входного символа и переход в состояние n (для терминалов) или просто переход в состояние n (для нетерминалов). Если n отрицательное, то выполняется свертка по правилу с номером $-n$. Также, для всех состояний, в которых есть свертки, выбирается *свертка по умолчанию* ($default\ reduce$). Такая свертка выполняется в случае ошибки, когда в состоянии нет действия, соответствующего входному символу.

6.1 Реализация

Приступим непосредственно к описанию реализации.

В *Bison*'е есть процедура, выполняющая построение таблицы $LALR$-транслятора. После вызова этой процедуры была добавлена новая процедура, которая выполняет анализ построенной таблицы по алгоритму, описанному в главе 4. А именно, она ищет все состояния транслятора, которые содержат ровно один перенос (не считая переноса символа конца строки) и не содержат сверток. Также для каждого такого состояния эта процедура запоминает проталкиваемый символ и номер состояния, в которое будет совершен переход.

Для того, чтобы получить результат данного анализа грамматики, *Bison* нужно вызывать с параметром *–report=states* или *–report=all*:

```
$ bison input.y --report=all
```

На грамматике, приведенной в предыдущей главе, 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
```

Анализ такого результата занимает много времени и требует определенных усилий (например, поиск состояний в таблице, анализ применяемых правил). Поэтому для удобства диагностики был реализован более подробный анализ состояний, показывающий все правила, используемые в состоянии:

```
state 0
    0 $accept: . E $end
    1 E: . E '+' E
    2    | . 'a'
    $default  push symbol 'a', and go to state 1
state 2
    0 $accept: E . $end
    1 E: E . '+' E
    $default  push symbol '+', and go to state 4
state 4
    1 E: . E '+' E
    1    | E '+' . E
    2    | . 'a'
    $default  push symbol 'a', and go to state 1
```

Но для больших грамматик такой вывод получается слишком объёмным. Поэтому подробные сообщения добавляются в файл диагностики построенного автомата аналогично сообщениям о свертках по умолчанию. В то же время на экран выводится краткая информация о наличии незначимых терминалов в состояниях. Это сделало инструмент более удобным.

6.2 Анализ грамматики ANSI C

Теперь попробуем проанализировать грамматику некоторого языка программирования. В качестве примера была выбрана грамматика *ANSI C* [6]. Полный результат анализа этой грамматики приведен в приложении А.

Рассмотрим наиболее интересные результаты.

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

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

В этом состоянии предполагается, что на входе будет открывающая скобка. Действительно, в этом случае мы можем опустить открывающую скобку. 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`. Данные ключевые слова являются стэйтментами, после них не может быть другого терминала. Т.е. если в тексте встретится слово `CONTINUE` или `BREAK`, то можно точно определить границы этого стэйтмента.

Самым интересным является цикл `DO...WHILE`. Правило, из которого он получается выглядит так:

```
iteration_statement -> DO statement WHILE '(' expression ')' ';'
```

Судя по результатам анализа, лишними в нем являются терминалы `WHILE`, `(`, `;` – состояния 178, 260 и 344 соответственно.

```
state 178
  196 iteration_statement: DO statement . WHILE '(' expression ')' ';'
    WHILE  shift, and go to state 260
    $default  push symbol WHILE, and go to state 260
state 260
  196 iteration_statement: DO statement WHILE . '(' expression ')' ';'
    '('  shift, and go to state 317
    $default  push symbol '(', and go to state 317
state 344
  196 iteration_statement: DO statement WHILE '(' expression ')' . ';'
    ';'  shift, and go to state 348
    $default  push symbol ';', and go to state 348
```

Действительно, после слова `DO` мы считываем стэйтмент, потом обязательно идет `WHILE (`, и в конце после закрывающей скобки – `;`. Символ `)` является обязательным из-за правила:

```
expression -> expression ',' assignment_expression
```

Т.е. в состоянии 259 может встретиться как и открывающая скобка, так и запятая:

```
state 259
   73 expression: expression . ',' assignment_expression
  195 iteration_statement: WHILE '(' expression . ')' statement
    ')'  shift, and go to state 316
    ','  shift, and go to state 226
```

Таким образом, цикл может быть представлен такой строчкой: “DO statement expression ')' ”. Наличие только закрывающей скобки выглядит не очень красиво. Справедливо заметить, что цикл вполне может

выглядеть так: “DO statement expression ’;’ ”. Этого можно достичь с помощью анализа пропущенных терминалов, описанного в главе 5.3.

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

```
state 94
  199 jump_statement: 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 при корректной входной строке только он может быть считан в этом состоянии. В данном случае, терминал имеет семантическое значение. Поэтому для того чтобы его можно было опустить, нужен некий конструктор, который должен быть написан программистом. Но в данном случае придумать какой-либо подходящий конструктор невозможно.

7 Заключение

В ходе данной работы были введены понятия ослабленного транслятора, синтаксически и семантически незначимых терминалов, избыточности грамматики с точки зрения построения $LALR$-трансляции. Был проведен обзор существующих алгоритмов исправления ошибок.

Основным результатом работы является предложение алгоритма исправления ошибок на основе анализа грамматики на избыточность. Главными преимуществами предложенного алгоритма являются следующие свойства:

- алгоритм не влияет на время трансляции корректных входных цепочек;
- для любой некорректной цепочки, которую может принять модифицированный транслятор, результат совпадает с результатом трансляции некоторой корректной цепочки;
- алгоритм автоматически строит ослабленный транслятор.

Также в ходе работы был реализован инструмент анализа грамматики на избыточность.

A Результаты анализа грамматики ANSI C

```
state 39 accepts only IDENTIFIER symbol;   default: push and go to 64
state 40 accepts only ')' symbol;   default: push and go to 67
state 63 accepts only IDENTIFIER symbol;   default: push and go to 64
state 88 accepts only ':' symbol;   default: push and go to 174
state 89 accepts only '(' symbol;   default: push and go to 175
state 90 accepts only '(' symbol;   default: push and go to 176
state 91 accepts only '(' symbol;   default: push and go to 177
state 93 accepts only '(' symbol;   default: push and go to 179
state 94 accepts only IDENTIFIER symbol;   default: push and go to 180
state 95 accepts only ';' symbol;   default: push and go to 181
state 96 accepts only ';' symbol;   default: push and go to 182
state 140 accepts only ')' symbol;   default: push and go to 238
state 148 accepts only ']' symbol;   default: push and go to 242
state 151 accepts only IDENTIFIER symbol;   default: push and go to 64
state 173 accepts only ':' symbol;   default: push and go to 255
state 178 accepts only WHILE symbol;   default: push and go to 260
state 180 accepts only ';' symbol;   default: push and go to 262
state 187 accepts only ')' symbol;   default: push and go to 268
state 188 accepts only IDENTIFIER symbol;   default: push and go to 269
state 193 accepts only IDENTIFIER symbol;   default: push and go to 274
state 241 accepts only IDENTIFIER symbol;   default: push and go to 307
state 254 accepts only ')' symbol;   default: push and go to 312
state 260 accepts only '(' symbol;   default: push and go to 317
state 298 accepts only ')' symbol;   default: push and go to 324
state 299 accepts only ')' symbol;   default: push and go to 325
state 301 accepts only ']' symbol;   default: push and go to 326
state 328 accepts only ')' symbol;   default: push and go to 341
state 330 accepts only ']' symbol;   default: push and go to 342
state 344 accepts only ';' symbol;   default: push and go to 348
total default pushes: 29
```

B Результаты анализа грамматики ANSI-ISO Pascal

```
state 1 accepts only IDENTIFIER symbol;   default: push and go to 8
state 2 accepts only IDENTIFIER symbol;   default: push and go to 8
state 5 accepts only SEMICOLON symbol;   default: push and go to 14
state 11 accepts only EQUAL symbol;   default: push and go to 19
state 16 accepts only IDENTIFIER symbol;   default: push and go to 8
state 20 accepts only IDENTIFIER symbol;   default: push and go to 8
state 21 accepts only DIGSEQ symbol;   default: push and go to 51
state 22 accepts only DOT symbol;   default: push and go to 54
state 26 accepts only EQUAL symbol;   default: push and go to 57
state 27 accepts only IDENTIFIER symbol;   default: push and go to 8
state 37 accepts only SEMICOLON symbol;   default: push and go to 14
state 59 accepts only SEMICOLON symbol;   default: push and go to 14
state 61 accepts only IDENTIFIER symbol;   default: push and go to 8
state 62 accepts only IDENTIFIER symbol;   default: push and go to 8
state 64 accepts only SEMICOLON symbol;   default: push and go to 14
state 67 accepts only SEMICOLON symbol;   default: push and go to 14
state 70 accepts only SEMICOLON symbol;   default: push and go to 14
state 71 accepts only SEMICOLON symbol;   default: push and go to 14
state 72 accepts only RPAREN symbol;   default: push and go to 137
state 97 accepts only IDENTIFIER symbol;   default: push and go to 8
state 99 accepts only DIGSEQ symbol;   default: push and go to 51
state 101 accepts only LBRAC symbol;   default: push and go to 145
state 104 accepts only IDENTIFIER symbol;   default: push and go to 8
state 106 accepts only OF symbol;   default: push and go to 148
state 109 accepts only OF symbol;   default: push and go to 154
state 110 accepts only IDENTIFIER symbol;   default: push and go to 8
state 111 accepts only DOTDOT symbol;   default: push and go to 157
state 114 accepts only SEMICOLON symbol;   default: push and go to 14
state 149 accepts only IDENTIFIER symbol;   default: push and go to 8
state 153 accepts only END symbol;   default: push and go to 202
state 163 accepts only IDENTIFIER symbol;   default: push and go to 8
state 164 accepts only COLON symbol;   default: push and go to 208
state 171 accepts only IDENTIFIER symbol;   default: push and go to 8
state 172 accepts only IDENTIFIER symbol;   default: push and go to 8
state 186 accepts only PBEGIN symbol;   default: push and go to 214
state 195 accepts only OF symbol;   default: push and go to 219
state 196 accepts only COLON symbol;   default: push and go to 220
```

```
state 208 accepts only IDENTIFIER symbol;   default: push and go to 8
state 211 accepts only IDENTIFIER symbol;   default: push and go to 8
state 217 accepts only OF symbol;   default: push and go to 260
state 220 accepts only IDENTIFIER symbol;   default: push and go to 8
state 223 accepts only END symbol;   default: push and go to 269
state 225 accepts only IDENTIFIER symbol;   default: push and go to 8
state 229 accepts only IDENTIFIER symbol;   default: push and go to 8
state 230 accepts only DIGSEQ symbol;   default: push and go to 51
state 234 accepts only IDENTIFIER symbol;   default: push and go to 8
state 235 accepts only COLON symbol;   default: push and go to 297
state 276 accepts only OF symbol;   default: push and go to 318
state 287 accepts only ASSIGNMENT symbol;   default: push and go to 324
state 290 accepts only THEN symbol;   default: push and go to 325
state 293 accepts only DO symbol;   default: push and go to 327
state 301 accepts only IDENTIFIER symbol;   default: push and go to 8
state 309 accepts only LPAREN symbol;   default: push and go to 343
state 315 accepts only RPAREN symbol;   default: push and go to 348
state 329 accepts only IDENTIFIER symbol;   default: push and go to 8
state 372 accepts only RPAREN symbol;   default: push and go to 388
state 394 accepts only DO symbol;   default: push and go to 402
state 399 accepts only RPAREN symbol;   default: push and go to 404
state 406 accepts only END symbol;   default: push and go to 409
total default pushes: 59
```

Список литературы

[1] *Кнут, Д.* Семантика языков программирования. Сборник статей / Д. Кнут, Ч. Хоор, П. Льюис. — Москва: Мир, 1980.

[2] *Мартыненко, Б.* Языки и трянсляции / Б. Мартыненко. — СПбГУ, 2004.

[3] *Терехов, А.* История и архитектура проекта RescueWare / А. Терехов, Л. Эрлих, А. Терехов // *Автоматизированный реинжиниринг программ.* — 2000. — Pp. 7–19.

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

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

[6] ANSI C grammar. — http://www.lysator.liu.se/c/ANSI-C-grammar-y.html.

[7] ANSI-ISO Pascal grammar. — http://www.moorecad.com/standardpascal/pascal.y.

[8] Bison - GNU parser generator. — http://www.gnu.org/software/bison/.

[9] *Burke, M.* A practical method for *LR* and *LL* syntactic error diagnosis and recovery / M. Burke, G. Fisher // *ACM Transactions on Programming Languages and Systems.* — April 1987. — Vol. 9. — Pp. 164–197.

[10] *Conway, R.* Design and implementation of a diagnostic compiler for PL/I / R. Conway, T. Wilcox // *Communications of the Association for Computing Machinery.* — March 1973. — Vol. 16. — Pp. 169–179.

[11] *Degano, P.* Comparison of syntactic error handling in LR parsers / P. Degano, C. Priami // *Software—Practice and Experience.* — June 1995. — Vol. 25. — Pp. 657–679.

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

[13] Relativity Technologies. — http://www.relativity.com.

[14] *Rohrich, J.* Methods for the automatic construction of error correcting parsers / J. Rohrich // *Acta Informatica.* — February 1980. — Vol. 13. — Pp. 115–139.

[15] Software Complexity. — http://www.stsc.hill.af.mil/crosstalk/1994/12/.