Санкт-Петербургский Государственный
Университет
Математико-механический факультет

Кафедра системного программирования

GLR-анализ

Курсовая работа студента 361 группы
Григорьева Семёна Вячеславовича

Научный руководитель ..... к.ф.-м.н. А.С. Лукичев
/подпись/

Санкт-Петербург
2009

Содержание

1 Введение.

Задачи автоматизированного реинжиниринга программ выдвигают особые требования к генераторам синтаксических анализаторов.

Для устаревшего языка сложно (а зачастую и невозможно) задать однозначную контекстно-свободную грамматику. Необходимо существенно преобразовать его спецификацию, которая приводится в документации, чтобы получить такую грамматику, но при этом она перестает быть сопровождаемой [1]. Поэтому устаревший язык обычно задается с помощью неоднозначной контекстно-свободной грамматики.

При поддержке нескольких диалектов языка необходима возможность лёгкой трансформации грамматики. Однако,зачастую, изменение одного правила приводит к появлению десятков конфликтов в грамматике [1], которые необходимо разрешать вручную. Это требует большого количества времени.

Как вариант решения этих задач предлагается использовать GLR грамматики и соответствующие инструменты построения анализаторов [1]. Действительно, GLR-алгоритм разрешает неоднозначности в грамматике на уровне концепции. По этому задание спецификации трансляции становится проще, требует меньше времени. Получившийся код компактнее и сопровождаемее.

Главным достоинством GLR-алгоритма является обработка неоднозначных грамматик. Анализатор, построенный по неоднозначной грамматике с помощью данного алгоритма, в результате разбора строит не единственное дерево, а несколько деревьев - лес, который можно сократить, используя специальные фильтры, а можно ,при задании в одной спецификации нескольких диалектов, вернуть весь лес для дальнейшего выбора нужного дерева/диалекта.

Работу алгоритма GLR можно рассматривать как параллельное исполнение набора LR-анализаторов. При этом данный набор дополняется процедурой управления стеками, оптимизирующей представление стеков путем их «склеивания» и «расклеивания», что позволяет хранить и строить параллельные выводы в рамках одного LR-анализатора, лишь в моменты их различия добавляя параллельный анализатор.

Оказалось, что весьма наглядно такой алгоритм может быть представлен в виде двух взаимно-рекурсивных функций (рекурсивно-восходящий алгоритм, recursive ascent). При этом расклеивание стека получается естественным образом как ветвление в одной из функций, а обратное склеивание может быть реализовано как кэширование результата функции.

Стоит отметить, что по производительности такой анализатор, являясь некоторой "надстройкой"над LR-анализатором, незначительно ему уступает. На сегодняшний день в соотношении производительность/класс разбираемых языков GLR-алгоритм выглядит наиболее предпочтительно.

Удобным способом формального определения грамматики, элементов и атрибутов языка программирования является расширенная нормальная форма Бэкуса-Наура. Поэтому инструмент должен работать с расширенными контекстно-свободными грамматиками.

При работе с инструментом пользователь ожидает получить результат описанный в терминах заданной им грамматики. Это выдвигает дополнительные требования к алгоритму. В случае, если входная грамматика была каким-либо образом преобразована, например с целью раскрыть конструкции EBNF, то появляется необходимость в построении "обратного"преобразования. Это преобразование должно "перевести"результат обратно в термины входной грамматики. Такие преобразования требуют дополнительных ресурсов и усложняют инструмент. Поэтому наиболее предпочтительными является алгоритмы, позволяющие обойтись без дополнительных преобразований грамматики.

2 Определения.

Определения используемых терминов:

- GLR:
- EBNF: Extended Backus-Naur Form – расширенная нормальная форма Бэкуса–Наура, расширенная БНФ. Способ формального определения грамматики, элементов и атрибутов языка программирования [3].
- ДКА:
- НДКА:
- Символ: общее название для терминала или нетерминал при описании КА.
- Замыкание: q* = q$\bigcup\{B \to .c|A \to a.Bb \in$q*$\}\bigcup\{x \to .x|A \to a.xb \in$q*$\}$
- Функция goto: goto q X = $\{A \to aX.b|A \to a.Xb \in$q*$\}$,

- LR-автомат:
- LR-ситуация:

3 Цели и задачи.

В рамках данной работы ставится цель разработки прототипа генератора анализаторов, удовлетворяющего основным перечисленным требованиям и предоставляющего, в то же время, ставшие привычными средства автоматизации трансляции (дополнение грамматики атрибутами, средства автоматического восстановления после ошибок, средства диагностики).

4 Обзор.

Предпочтительным алгоритмам анализа является алгоритм разбора произвольной контекстно-свободной грамматики. Поэтому нужно рассмотреть инструменты, основанные на этом алгоритме. Важен способ реализации алгоритма, так как существуют несколько альтернатив: алгоритм Томиты (GLR-алгоритм), алгоритм Эрли (Early), рекурсивно-восходящий алгоритм. Так-же следует обратить преобразования грамматики, необходимые для построения анализатора.

В настоящее время существуют следующие инструменты основанные на GLR-алгоритме.

- ASF+SDF [11] (Algebraic Specification Formalism + Syntax Definition Formalism) - генератор с широкими возможностями, но достаточно сложным входным языком. Является SGLR-инструментом (Scannerless, Generalized-LR).
- Bison [12] - развитие инструмента YACC. Все грамматики, созданные для оригинального YACC, будут работать и в Bison. Является одним из самых популярных и совершенных "потомков"YACC. При включении соответствующей опции использует GLR-алгоритм (по умолчанию LALR).
- Elkhound [13] - позиционируется как быстрый и удобный GLR-инструмент, созданный в университете Беркли (США), тем не менее обладает достаточно "бедным"входным языком (например, он не поддерживает конструкций расширенной формы Бэкуса-Наура).

В работе [4] проведён подробный анализ этих инструментов.

Jade это генератор рекурсивно-восходящих LALR(1) парсеров с целевым языком C. Его подробное описание приводится в статье [7]. При реализации данного инструмента появилась проблема объёма кода целевого парсера. Так как при построении детерминированного парсера необходимо генерировать процедуры для каждого состояния, то объём кода быстро растёт, с ростом количества правил в грамматике. Так для языка Java объём кода составляет примерно 4 мегабайта [7]. В Jade эта проблема решается путём создания глобальной структуры(массива состояний), где хранится информация, позволяющая переиспользовать процедуры [7].

5 Реализация.

Платформой для создания инструмента выбран .NET . Основным языком реализации является F# [2]. Такой выбор сделан потому, что F# является функциональным языком программирования. Это позволяет проще работать с такими структурами данных как списки и деревья, которые неизбежно возникают при решении поставленной задачи.

Общая схема реализуемого алгоритма такова:

- Входные данные – грамматика в виде списка правил. Одно правило - специальная структуры данных, описанная ниже;
- Построение ДКА по правой части правила.
- Построение LR-автомата;
- Построение деревьев вывода;

5.1 Внутреннее представление.

За основу внутреннего представления грамматики взято представление инструмента YARD. Это представление и ,соответственно, входной язык инструмента, содержит такие конструкции, как

- Конструкции расширенной формы Бэкуса-Наура;
- Макроправила (параметризация одних правил другими);
- Сгруппированные альтернативы;
- Предикаты;

5.2 Рекурсивно восходящий алгоритм.

Существует альтернатива табличным анализаторам - рекурсивно-восходящие анализаторы. Идея состоит в том, чтобы не использовать стек явно, а заменить его стеком вызова рекурсивных функций и эмулировать поведения автомата вызовом функций. Для этого можно построить функцию соответствующую каждому состоянию и получить детерминированный LR анализ [9].

Таким образом - рекурсивно восходящий алгоритм это аналог рекурсивного спуска, но для LR грамматики. Однако, при такой реализации возникают проблемы с быстрым ростом объёма кода целевых функций [7].

Существует подход к решению этой задачи несколько с другой стороны. Показано, что кожно построить систему функций, оперирующие уже не одним состоянием, а множеством состояний, и свести ,таким образом, количество необходимых функций к двум взаимно рекурсивным [10]:

- parse q i $=\{(A \to a., i) | A \to a. \in q\} \bigcup$
  $\{(A \to a.b, k) | i = xj, (A \to a.b, k) \in climb$ q x j$\} \bigcup$
  $\{(A \to a.b, k) | B \to e, (A \to a.b, k) \in climb$ q B j $\}$
- climb q X i $= \{(A \to a.Xb, k) | (A \to aX.b, k) \in parse($goto q X$)i, a \neq e, A \to a.Xb \in q\} \bigcup$
  $\{(A \to a.b, l) | (C \to X.c, j) \in parse($goto q X$)i, (A \to a.b, l) \in climb$ q C j$\}$

Пользуясь таким определением можно построить функции из статьи [9], однако сейчас интереснее их реализация, согласно определению.

Важно, что при решении нашей задачи нет необходимости генерации функции для каждого состояния. Так как цель - получение недетерминированного разбора, то можно реализовать всего две функции: parse и climb, определённые выше. Это позволит решить проблему объёма кода, возникшую в Jade. Объём кода резко сократится и, более того, будет получен недетерминированный вариант разбора.

Используя рекурсивно-восходящий алгоритм можно получить эквивалент алгоритму Томиты. При использовать описанных выше функции автоматически получается ветвление в момент возникновения неоднозначности (разделение стека в алгоритме Томиты). Это получается благодаря тому, что они принимают на вход состояние и возвращают множество всех возможных состояний.

Однако возникает проблема - сложность алгоритма, основанного на функциях parse и climb экспоненциальная. Для борьбы с этим в статье [8] предлагается дополнить функции механизмом запоминания результатов предыдущих вызовов, чтобы при очередном вызове , в случае, если эта функция уже вызывалась с такими параметрами, то результат возвращался без повторного вычисления. Доказывается [8], что в при такой реализации оценки по времени будут порядка $O(n^3)$.

Запоминание результатов предыдущих вычислений и будет аналогом объединения состояний и получения структурированного в виде графа стека (graph-structured stack) в алгоритме Томиты.

Технически этого можно добиться реализовав функцию memoize. На языке F# её можно реализовать так:

```
let memoize (f: 'a ->'b) =
     let t = new System.Collections.Generic.Dictionary<'a,'b>()
     in
     fun a ->
            if t.ContainsKey(a)
            then t.[a]
            else
               let res = f a
               in
               t.Add(a,res);
               res
```

В дальнейшем в качестве функции f будет выступать функция parse и climb.

Дополнительного ускорения можно добиться заранее вычислив функцию goto. Действительно, построение замыкания (closure), необходимое для вычисления функции goto дорогая операция, а вызов goto происходит при каждом вызове функции climb. Поэтому можно вычислить goto на этапе построения анализатора, и в целевой программе (в функции climb) свести вызов (goto q X) к поиску по ключу.

С учётом этих дополнений на F# функции `parse` и `climb` могут быть реализованы следующим образом:

```
let rec climb =
    memoize (fun (states,(symbol,i)) ->
    if states = Set.empty
    then Set.empty
    else
```

```
    let gt =  goto (states,symbol)
    let new_states = parse (gt,i)
    if Set.exists (fun ((state,tree),i)->
                       state.prod_name="S"
                        &&
                       state.next_num=None&&i=1)
                       new_states
    then states
    else
    Set.union_all
    [Set.filter (fun items->
                  Set.exists (fun item  ->
                                  (exists ((=) (fst items)) (nextItem item) )
                                   &&
                                  (item.item_num <> item.s)
                               )states)new_states
     |>(Set.map (fun items->
                     (map (fun itm ->
                               (itm,snd items))
                          (prevItem (fst items)))))|>union_all

    ;
    Set.union_all(
    union_from_Some[for (item,i) in new_states ->
                    if (exists (fun itm -> (getText itm.symb) = x)
                               (prevItem item))
                        &&
                       (item.prod_name<>"S")
                        &&
                       (exists (fun itm -> itm.item_num=item.s)(prevItem item))
                    then Some(climb (states,item.prod_name,i))
                    else None])
    ])
and parse =
    memoize (fun (states,i) ->
    union_all
        [map (fun state -> (state,i))
             (Set.filter (fun item -> (item.next_num=None))states)
         ;if (get_next_lex i = m_end)
          then empty
          else climb(states,mgetText(get_next_lex i),i-1)
```

```
])
```

5.3 Расширенные контекстно-свободные грамматики.

На практике оказывается удобно пользоваться расширенной контекстно-свободной или EBNF грамматикой. То-есть такой грамматикой, у которой в правых частях правил могут использоваться конструкции регулярных выражений. Необходимость использования EBNF в инструментах генерации анализаторов обоснована в работе [4].

Поддержка регулярных выражений в правых частях правил (EBNF-грамматики) получается естественным образом. Для этого правая часть правила представляется как конечный автомат. LR-ситуация в таком случае может быть представлена парой: правило (нетерминал+КА) и номер состояния (соответствует позиции маркера в классическом определении). Проблемы определения левой границы отрезка в магазине, соответствующего текущему правилу, в данном подходе не существует, так как стек вызовов рекурсивных функций хранит информацию о начале анализа по правилу.

В качестве примера рассмотрим грамматику арифмитических выражений без приоритетов :

1) $E \rightarrow E$'+'E
2) $E \rightarrow E$'$*$'E
3) $E \rightarrow$'('E')'
4) $E \rightarrow$'a'

Так-как будет необходимо строить LR-фвтомат, то пополним грамматику. Добавим стартовый нетерминал и ещё одно правило.

1) $S \rightarrow E$

Грамматика примет вид:

1) $S \rightarrow E$
1) $E \rightarrow E$'+'E
2) $E \rightarrow E$'$*$'E
3) $E \rightarrow$'('E')'
4) $E \rightarrow$'a'

В данном случае удобно объединить правила 1) и 2) в одно:

$E \rightarrow E$('+' | '$*$')E

Далее:

$E \rightarrow (E$('+' | '$*$')$E)|($ '('E')' $)|$'a' (1).

Используя конструкции EBNF мы смогли уменьшить количество правил в грамматике до двух:

$E \to E(\text{'+'} \mid \text{'*'})E$
$E \to (E(\text{'+'} \mid \text{'*'})E) \mid (\ \text{'('}E\text{')'}\ ) \mid \text{'}a\text{'}$

5.3.1 Построение ДКА

Для того, чтобы построить табличный LR анализатор необходимо преобразовать расширенную контекстно-свободную грамматику. Нужно избавиться от конструкции BNF в правых частях. Однако можно попробовать обойтись без преобразований грамматики.

Существуют способы поддержки расширенной контекстно-свободной грамматики на уровне анализатора, без преобразований входной грамматики [8]. Для этого нужно переопределить функцию goto. Можно заменить позицию точки в правиле на номер состояния конечного автомата, построенного по соответствующему регулярному выражению. Заметим, что определённая выше функция goto может быть описана в терминах конечного автомата. В этом случае регулярное выражение и ,следовательно ,построенный по нему автомат очень прост.

Для построения конечного автомата по регулярному выражению воспользуемся алгоритмом Томпсона [6]. Результатом работы данного алгоритма является недетерминированный конечный автомат (НКА). Так как правая часть правила (1) - регулярное выражение, то применим к ней алгоритм Томпсона. Получим следующий НКА: **здесь должен быть граф**

Каждое состояние и каждый переход КА должны преобразоваться в состояние и переход LR автомата соответственно. Заметим, что в в полученном НКА много ε-переходов. Например, каждая альтернатива вносит два дополнительных состояния и 4 ε-перехода. Чтобы уменьшить количество переходов в результирующем LR-автомате можно преобразовать НКА в детерминированный конечный автомат(ДКА). Для этого применим стандартный алгоритм преобразования НКА в ДКА [6]. Полученный в результате преобразований ДКА будет выглядеть следующим образом:**здесь должен быть граф**

На практике удобно представить результирующий ДКА в виде тройки (S,F,R), где S – начальное состояние ДКА , F – множество конечных состояний и R – множество фугкций перехода (правил), задающее ДКА.

5.3.2 Построение множества ситуаций

Перейдём к построению LR ситуаций. Как было сказано выше, основная идея состоит в том, что точку из классического определения ситуации нужно заменить состоянием ДКА, построенног опо правой

части проавила, как описано выше. Поясним. Предположим, что правило грамматики не содержит конструкций BNF. Например:

$E \to E$'+'E

Множество ситуаций для данного правила:

1) $E \to .E$'+'E
2) $E \to E$'.+'E
3) $E \to E$'+'$.E$
4) $E \to E$'+'$E.$

Теперь построим ДКА в соответствии с нашим алгоритмом. Результат показан на рисунке ниже.

Рис. 1:

У построенного ДКА четыре состояния: S_1 , S_2 , S_3 , S_4. Сопоставим состоянию S_1 ситуацию $E \to .E$'+'E, S2 ситуацию $E \to E$'.+'E и так далее. В итоге получим соответствие:

S_1 - $E \to .E$'+'E
S_2 - $E \to E$.'+'E
S_3 - $E \to E$'+'$.E$
S_4 - $E \to E$'+'$E.$

Таким образом, каждой LR-ситуации мы смогли сопоставить состояние ДКА, построенного по правой части правила. Заметим, что ситуация характеризуется правилом, для которого она строится и положением точки в правой части этого правила. Теперь вернёмся к предыдущему примеру и обобщим ситуацию.

В правой части правила - регулярное выражение. Выше мы построили по нему ДКА. Теперь построим для каждого состояния ДКА LR-ситуацию.

Для этого сперва опишем структуру данных, в которой будем сохранять одну ситуацию. Ситуация строится для правила грамматики. Правило – это нетерминал в левой части и ДКА, построенный по правой части. ДКА задаётся стартовым состоянием, множеством правил перехода и множеством конечных состояний. Правило перехода для ДКА – номер текущего состояния, принимаемый символ, номер состояния, в которое переходит ДКА, приняв данный символ. Всю эту информацию необходимо хранить, для дальнейшей работы с ней. В итоге для этого потребуется структура со следующими полями:

- Номер правила.
- Левая часть правила(нетерминал).
- Номер текущего состояния ДКА.
- Символ принимаемый ДКА в данном состоянии.
- Номер состояния ДКА, в которое он перйдёт приняв данный символ.
- Номер начального состояния ДКА.
- Номера конечных состояний ДКА.

Заметим, что эта информация состоит из двух частей: ниформации о ДКА в целом и правило перехода. Это ляжет в онову функции построения множества ситуаций по ДКА.

```
create_items (DFA |
(DFA построен для правила с номером = prd_num и нетерминалом в левой части = pr
 && (DFA = s,f,{rule | rule = (cur_num,symb,next_num)})) =
{(prod_number,prod_name,item_number,symbol,next_number,start_state,finale_state
  prod_number = prd_num, prod_name = prd_name, item_number = cur_num,
  symbol = symb, next_number = next_num,start_state = s, finale_state = f}
```

5.3.3 Вычисление GOTO

Вычисление функции GOTO - очень дорогая операция. При анализе вызов происходит при каждом вызове функции parse. По этому вычисление GOTO каждый раз сильно ухудшит производительность. Самое простое решение этой проблемы - вычислить значеня функции заранее, на стадии генерации данных.

На данном этапе проще всего вычислить GOTO для всех возможных пар (состояние, символ) и сконструировать коллекцию, в которой , вместо вычислений, будет производиться поиск при анализе. В качестве ключей можно взять, например, значение hash-функции от значения параметров, для которых вычислено соответствующее значение.

Основа алгоритма - стандартный алгоритм вычисления GOTO при LR анализе, подробное описание которого можно найти в **ссылка**. Небольшое изменение лишь в том, что ситуация имеет специальный вид.

5.4 Построение деревьев вывода.

Для практических целей одна только информация о том, принадлежит входная цепочка данному языку или нет, не очень полезна. Гораздо более ценный результат - дерево вывода цепочки в данной грамматике. Так как обсуждается недетерминированый алгоритм, то в нашем случае речь будет идти о множестве деревьев (лесе) вывода. Для краткости будем называть его лесом вывода.

Рекурсивно-восходящий алгоритм анализа, описанный выше, будет основой следующего шага. Теперь необходимо получить деревья вывода.

Будем строить дерево сразу во время анализа. Для этого потребуется изменить функции `parse`и `climb`,описанные ранее.

Так как инструмент предназначен для работы с неоднозначными грамматиками, то в общем случае он должен строить лес вывода. То есть в случае если грамматика неоднозначна и существует несколько способов вывода некоторой входной цепочки, то должны быть построены все возможные деревья вывода.

Вначале рассмотрим детерминированный случай, а затем перейдём к недетерменированному.

При детерминированном разборе, если цепочка пораждается входной грамматикой, должно быть получено единственное дерево вывода.

На данном этапе структуру дерева можно сделать минимально простой. Для каждой вершины можно хранить только имя сооветствущего нетерминала или теоминала и информацию о сыновьях. Для представления дерева опишим соответствующий алгебраический тип:

```
type tree = Node of string*tree*tree|Leaf of string
```

Введём следующие обозначения: $A \to R$ – правило грамматики, где A - нетерминал, R - ДКА, построенный по регулярному выражнению; $(A \to R, i)$ – LR-ситуация, где i - состояние ДКА; $is - final(R, i)$ – проверка, что i - конечное состояние R; q - LR-состояние (core); $q*$ - замыкание q; $(leaf : a)$, $(A- > ...)$ – конструкции дерева разбора

Тогда сигнатуры функции будут иметь следующий вид:

$parse\ q\ \{u|A \to a.b, u = vx, b \to v\} \to (A \to a.b)x\{tree|tree-$синтаксическое дерево вывода для $t \in b\})$

$climb\ q\ X\ \{u|A \to aX.b, u = vx, b \to v\}(tree$:синтаксическое дерево для X) $\to (A \to a.Xb)x\{tree|tree-$синтаксическое дерево вывода для $t \in Xb\}$

А сами функции будут выглядеть так:

```
parse q u =
```

```
    if exists (b A → R,i) q & is-final(R,i)
    then (A → R,i;u;[])
    else
      if u=av & exists ( A →  R,i) q*  &  R(i,a)=j
      then climb q a v (leaf:a)
    else
    if exist (A → R,0) q* & is-final(R,0)
    then climb q A v (A → [])
climb q X u h =
     let (A → R,j;w;s) = parse (goto q* X) u in
     if R(i,X)=j & (A→  R,i) in q
     then (A →  R,i;w;h::s)
    else climb q A w (A →  h::s)
```

Теперь нужно обобщить эти функции для случая, когда возможно несколько деревьев вывода. Это можно сделать следующим образом. Заметим, что функции `parse` и `climb` для разбора произвольной грамматики получают в качестве параметра не одно состояние, а множество состояний. Ясно, что новые функци должны, в сущности, работать так же как и функции для детерминированного разбора, только обрабатывать не одну пару `(state,trees)` , где state - состояние и trees - множество деревьев вывода, соответствующих этому состоянию, а множество таких пар. Таким образом, достаточно заменить состояние парой `(state,trees)` в функциях описанных в п. 3.2.

Пронумеруем правила грамматики.

1) $S \to E$
2) $E \to E + E$
3) $E \to E * E$
4) $E \to (E)$
5) $E \to a$

Данная грамматика порождает арифметические выражения с двумя бинарными операциями и скобками.

Очевидно, что такая грамматика неоднозначна. Существуют цепочки, которые выводимы несколькими способами. Например: возьмём цепочку a+a+a. Она имеет два левосторонних вывода:

$S \xrightarrow{1} E \xrightarrow{2} E + E \xrightarrow{5} a + E \xrightarrow{2} a + E + E \xrightarrow{5} a + a + E \xrightarrow{5} a + a + a$

и

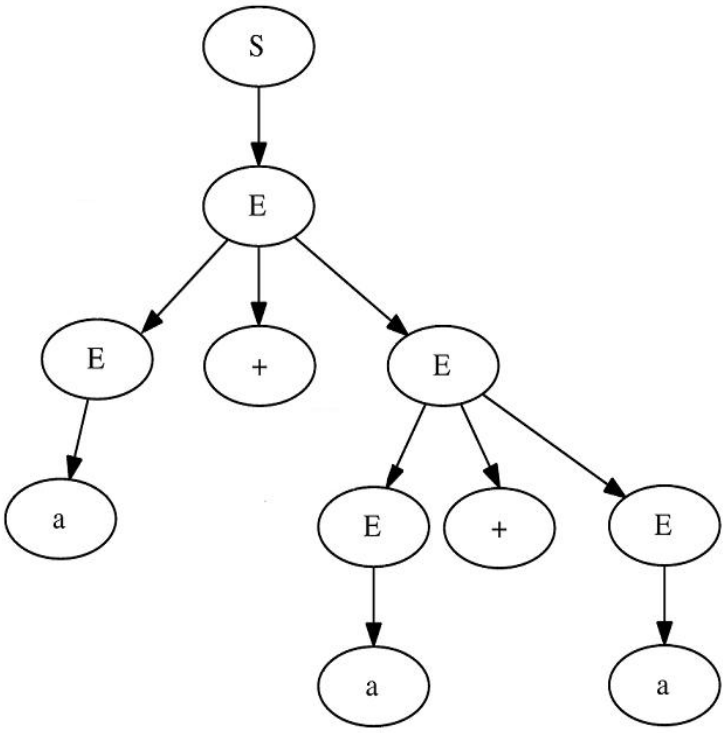
$S \xrightarrow{1} E \xrightarrow{2} E + E \xrightarrow{2} E + E + E \xrightarrow{5} a + E + E \xrightarrow{5} a + a + E \xrightarrow{5} a + a + a$

Над стрелкой - номер применяемого на данном шаге вывода правила.

Видим, что существует два вывода цепочки в данной граматике:

(1) → (2) → (5) → (2) → (5) → (5)

(1) → (2) → (2) → (5) → (5) → (5)
Им соотвтствуют деревия вывода:



Соответ

6 Заключение.

Полученные результаты:

- проведён обзор инструментов поатроения анализаторов. Выяснено, что предпочтительным алгоритмом анализа является GLR алгоритм;
- изучен рекурсивно-восходящий алгоритм анализа;
- предлжен прототип инструмента реализующего рекурсивно-восходящий алгоритм и применимого для работы с неоднозначными грамматиками;

6.1 Свойства прототипа.

Предложеный прототип имеет следующие характеристики:

- по однозначной LR-грамматике строится анализатор с линейной сложностью;
- по неоднозначной грамматике строится анализатор, возвращающий все возможные деревья вывода для данной входной цепочки;
- показана возможность поддержки регулярных выражений в правых частях правил;

6.2 Дальнейшее развитие.

Задачи, требующие решения:

- поддержка атрибутных грамматик;
- автоматическое востановление после ошибок;

7 Приложения

7.1 Внутреннее представления грамматики в инструменте YARD

Ниже приведено описание структуры данных, соответствующее внутренниму представлению грамматики в инструменте YARD, на языке F# .

```
module Definition =
  struct
    type ('patt,'expr) t =
    {
     head    :'expr option;
     grammar : Grammar.t<'patt,'expr>;
     foot    :'expr option
    }
end
```

Модуль `Definition` соответствует фвйлу с описанием грамматики. Элементы `head` и `foot` соответствуют длополнительным описаниям в

начале и в конце файла соответсвенно. Например, подключение модулей, какие-либо дополнительные действия, которые должны быть включены в целевой код „как есть“.

```
module Grammar =
  struct
    type t <'patt,'expr> = (Rule.t<'patt,'expr>) list
  end
```

Модуль `Grammar` – представление грамматики. Грамматика это список правил.

```
module Rule =
  struct
    type t <'patt,'expr> =
    {
     name    : string;
     args    : 'patt list;
     body    : (Production.t <'patt,'expr>);
     _public : bool;
     metaArgs: 'patt list
    }
  end
```

Модуль `Rule` – представление правила грамматики. Правило имеет имя. Оно может применяться к аргуметтам или мета аргументам (`adgs` и `metaArgs` соответственно). Более подробно об этом можно прочеть в работе [4]. Элемент `_public` указывает, является ли правило стартовым. В общем случае таких правил в грамматике может быть несколько. Элемент `body` – правая часть правила(продукция).

```
module Production =
  struct
    type elem <'patt,'expr> =
    {
     omit:bool;
     rule:(t<'patt,'expr>);
     binding:'patt option;
     checker:'expr option
    }
    and
    t <'patt,'expr> =
```

```
    |PAlt     of (t <'patt,'expr>) * (t<'patt,'expr>)
    |PSeq     of (elem <'patt,'expr>) list * 'expr option
    |PToken   of Source.t
    |PRef     of Source.t * 'expr option // Vanilla rule reference with an optio
    |PMany    of (t <'patt,'expr>) //expr*
    |PMetaRef of Source.t * 'expr option * 'expr list // Metarule reference like
    |PLiteral of Source.t
    |PRepet   of (t <'patt,'expr>) * int option * int option  //extended regexp
    |PPerm    of (t <'patt,'expr>) list //permutation (A || B || C)
    |PSome    of (t <'patt,'expr>) //expr+
    |POpt     of (t <'patt,'expr>) //expr?

  end
```

Модуль `Production`– продукция.

```
module Source =
  struct
   type t = string * (int * int)

   let toString ((r,_):t):string = r

  end
```

Список литературы

[1] *Mark G.J. van den Brand, Alex Sellink, Chris Verhoef* Current Parsing Techniques in Software Renovation Considered Harmful.// IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension. - IEEE Computer Society, Washington,1998.

[2] *h*ttp://www.research.microsoft.com/fsharp (дистрибутивы и документация по языку F#)

[3] *I*SO/IEC 14977 : 1996(E)

[4] *Чемоданов И.С.* Генератор синтаксических анализаторов для решения задач автоматизированного реинжиниринга программ.

[5] *Dick Grune, Ceriel Jacobs* PARSING TECHNIQUES A Practical Guide

[6] *Ахо А., Сети Р., Ульман Дж.* Компиляторы: принципы, технологии, инструменты. М:. Издательский дом <Вильямс>2003. 768 с.

[7] *Ronald Veldena* Jade, a recursive ascent LALR(1) parser generator. September 8,1998

[8] *Rene Leermakers* Non-deterministic Recursive Ascent Parsing. Philips Research Laboratories, P.O. Box 80.000, 5600 JA Eindhoven, The Netherlands.

[9] *Larry Morell, David Middleton* RECURSIVE-ASCENT PARSING. Arkansas Tech University Russellville, Arkansas.

[10] *Lex Augusteijn* Recursive Ascent Parsing (Re: Parsing techniques). lex@prl.philips.nl (Lex Augusteijn) Mon, 10 May 1993 07:03:39 GMT

[11] http://www.cwi.nl/projects/MetaEnv (сайт разработчиков ASF+SDF)

[12] http://www.gnu.org/software/bison (сайт разработчиков Bison)

[13] http://www.cs.berkeley.edu/smcpeak/elkhound (сайт разработчиков Elkhound)