



Министерство науки и высшего образования Российской
Федерации Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Московский государственный технический
университет имени Н.Э. Баумана
(национальный исследовательский
университет)» (МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Теоретическая информатика и компьютерные
технологии

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ
РАБОТЕ**

ПО КУРСУ КОНСТРУИРОВАНИЕ КОМПИЛЯТОРОВ

НА ТЕМУ:

EDSL для Python для написания лексического и
синтаксического

анализа

Студент

подпись, дата

фамилия, и.о.

Научный руководитель

подпись, дата

фамилия, и.о.

Оглавление	
ВВЕДЕНИЕ	3
ТЕОРЕТИЧЕСКАЯ ЧАСТЬ	4
Обозначения и основные понятия.....	4
Общее описание алгоритма.....	4
LR-анализ.....	7
Построение LR(0)-автомата.....	11
Таблицы ACTION и GOTO.....	12
Поведение LR-анализатора.....	12
LALR-анализатор.....	13
Пункты и переопределение процедур.....	13
Канонические таблицы LR(1)-анализа.....	14
Построение LALR-таблиц синтаксического анализа.....	15
РЕАЛИЗАЦИЯ	21
Входные данные.....	21
Терминалы.....	21
Нетерминальные символы и правила грамматики.....	21
Лексический анализ.....	23
Синтаксический анализ.....	24
Построение LALR(1)-таблицы.....	24
Парсер.....	25
ТЕСТИРОВАНИЕ	27
ЗАКЛЮЧЕНИЕ	28

ВВЕДЕНИЕ

Целью данной курсовой работы является написание собственной библиотеки на Python для генерации парсера по заданной грамматике.

Грамматика описывается с помощью средств этой библиотеки и имеет понятный синтаксис, что позволит пользователю упростить работу с контекстно-свободной грамматикой. Каждое правило грамматики может ассоциироваться с семантическим правилом, которое будет выполняться, если парсер встретит заданное семантическое правило. Подобный парсер управляется таблицей, поэтому в работе осуществляется построение LALR-таблицы.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Обозначения и основные понятия

Будем рассматривать контекстно-свободную грамматику или просто грамматику как:

1. Множество терминальных символов. Элементарные символы языка, определяемые грамматикой.
2. Множество нетерминалов. Нетерминал представляет собой множество строк терминалов.
3. Множество продукций, каждая из которых состоит из нетерминала в левой части продукции, стрелки (или другого условного символа) и последовательности терминалов и/или нетерминалов, называемых телом или правой частью продукции.
4. Стартовый символ, принадлежащий нетерминальным символам.

Общее описание алгоритма

Лексический анализатор считывает символы исходной программы и группирует их в лексически осмысленные единицы – лексемы и возвращает токены, представляющие эти лексемы. Токен имеет два значения – имя токена и значения атрибута. Имена токенов называются терминалами, т.к. они появляются в грамматике языка в виде терминальных символов. С помощью перегрузки операторов можно описать грамматику непосредственно в тексте программы.

Для синтаксического анализа был использован алгоритм LALR(1). Восходящий синтаксический анализатор основан на концепции LR(k)-анализа. L означает сканирование входного потока слева направо, R – построение правого порождения в обратном порядке, k – количество предпротматриваемых символов входного потока, необходимое для принятия решений.

LR-анализаторы управляются таблицами, а грамматика, для которой можно построить таблицу синтаксического анализа соответствующими способами, называется LR-грамматикой. Чтобы грамматика была LR, достаточно, чтобы синтаксический анализатор, работающий слева направо методом переноса свертки, был способен распознавать основы правосентенциальных форм при их появлении на вершине стека.

У такого анализатора есть ряд преимуществ:

1. Они могут быть созданы для распознавания практически всех конструкций языков программирования, для которых может быть написана контекстно-свободная грамматика. Существуют кс-грамматики, не являющиеся LR, но в большинстве случаев для типичных конструкций яп их можно избежать.
2. LR-анализатор может обнаруживать синтаксические ошибки сразу же, как только это становится возможным при сканировании входного потока.
3. Класс грамматик которые могут быть проанализованы с использованием LR-методов представляет собой надмножество класса грамматик, которые могут быть проанализированы с использованием предиктивных или LL-методов синтаксического анализа. В случае LR(k)-грамматик должна быть возможность распознать правую часть продукции в порожденной ею правосентенциальной форме с дополнительным предпросмотром k входных символов. Это требование существенно мягче требования для LL(k)-грамматик, в которых необходимо распознать продукцию по первым k символам порождения ее тела.

В программе создается восходящий lookahead-парсер (т.е. синтаксический анализ типа «перенос-свертка»). Для построения восходящего парсера необходимо построения свертки: на каждом шаге свертки некоторая

подстрока, соответствующая телу продукции, заменяется нетерминалом в заголовке этого правила. У парсера типа перенос-свертка есть 4 возможных действия:

1. Перенос. Переносит следующий символ входной строки на вершину стека.
2. Свертка. Правый конец строки свертки находится на вершине стека. Необходимо найти левый конец строки и решить каким нетерминалом заменить строку.
3. Ассерпт. Парсинг успешно завершен.
4. Ошибка. Синтаксическая ошибка и вызов ошибки

В синтаксическом анализе перенос-свертка для хранения символов грамматики используется стек. Стек выбран не случайно: основа всегда появляется на вершине стек, а не внутри него. Для хранения непроанализированной части входной строки используется входной буфер. Для маркировки дна стека и правого конца строки используется символ \$. Изначально стек пуст, а во входном буфере находится исходная строка. Рисунок 1 иллюстрирует описанную модель.

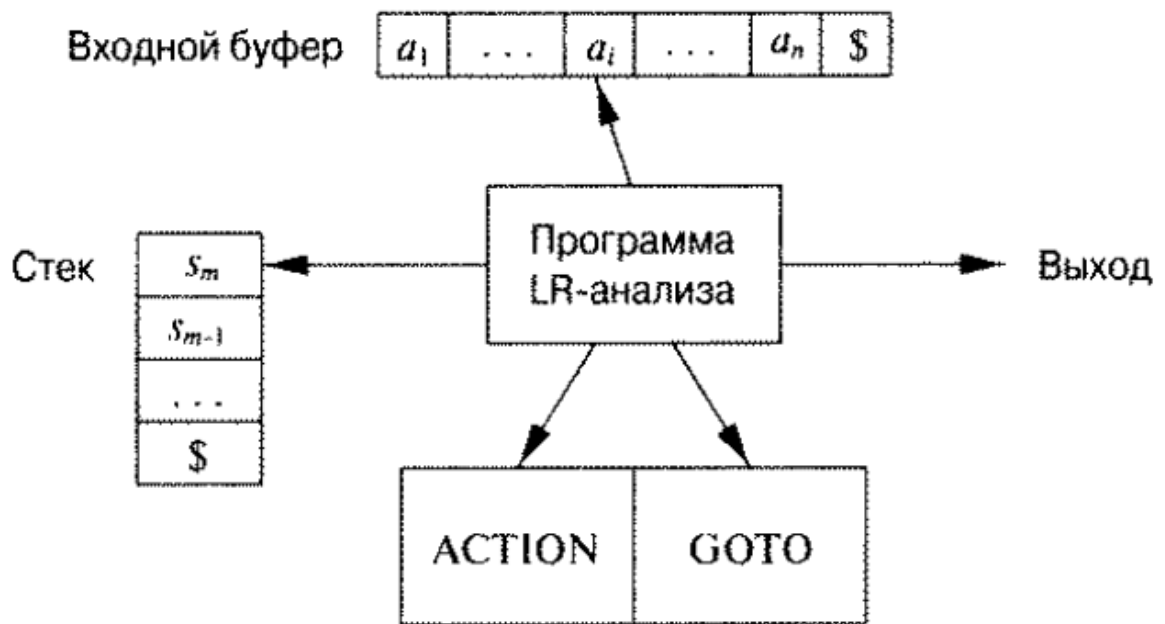


Рисунок 1 - Модель синтаксического анализатора, управляемого таблицей

В процессе сканирования входной строки слева направо синтаксический анализатор выполняет переносы символов на стек, пока не будет готов выполнить свертку строки n символов грамматики на вершине стека. Затем выполняется свертка к заголовку соответствующей продукции. Это повторяется до тех пор, пока не будет обнаружена ошибка или пока стек не будет содержать только стартовый символ, а входной буфер не содержать символов.

LR-анализ

Построение LR(0)-автомата

ПС-анализатору необходимо принимать решение о выборе операции – переноса или свертки, т.е. он должен знать, что на вершине стека не основа и корректное действие перенос, либо наоборот.

Пункты

Для этого анализатору необходимо поддерживать состояния, которые отслеживают, где именно в процессе синтаксического анализа мы находимся. Состояния представляют собой множества пунктов. LR(0)-пункт грамматики

G – это продукция G с точкой в некоторой позиции правой части. Например, продукция $A \rightarrow XY$ имеет три пункта.

Таким образом, пункт указывает, какую часть продукции уже рассмотрели в процессе синтаксического анализа.

Один набор множеств LR(0)-пунктов, именуемый каноническим набором LR(0), обеспечивает основу для построения детерминированного конечного автомата или LR(0)-автомата.

Каждое состояние LR(0) автомата представляет собой множество пунктов в каноническом наборе LR(0).

В качестве примера можно рассмотреть грамматику арифметических выражений:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

LR(0)-автомат для данной грамматики представлен на рисунке 2.

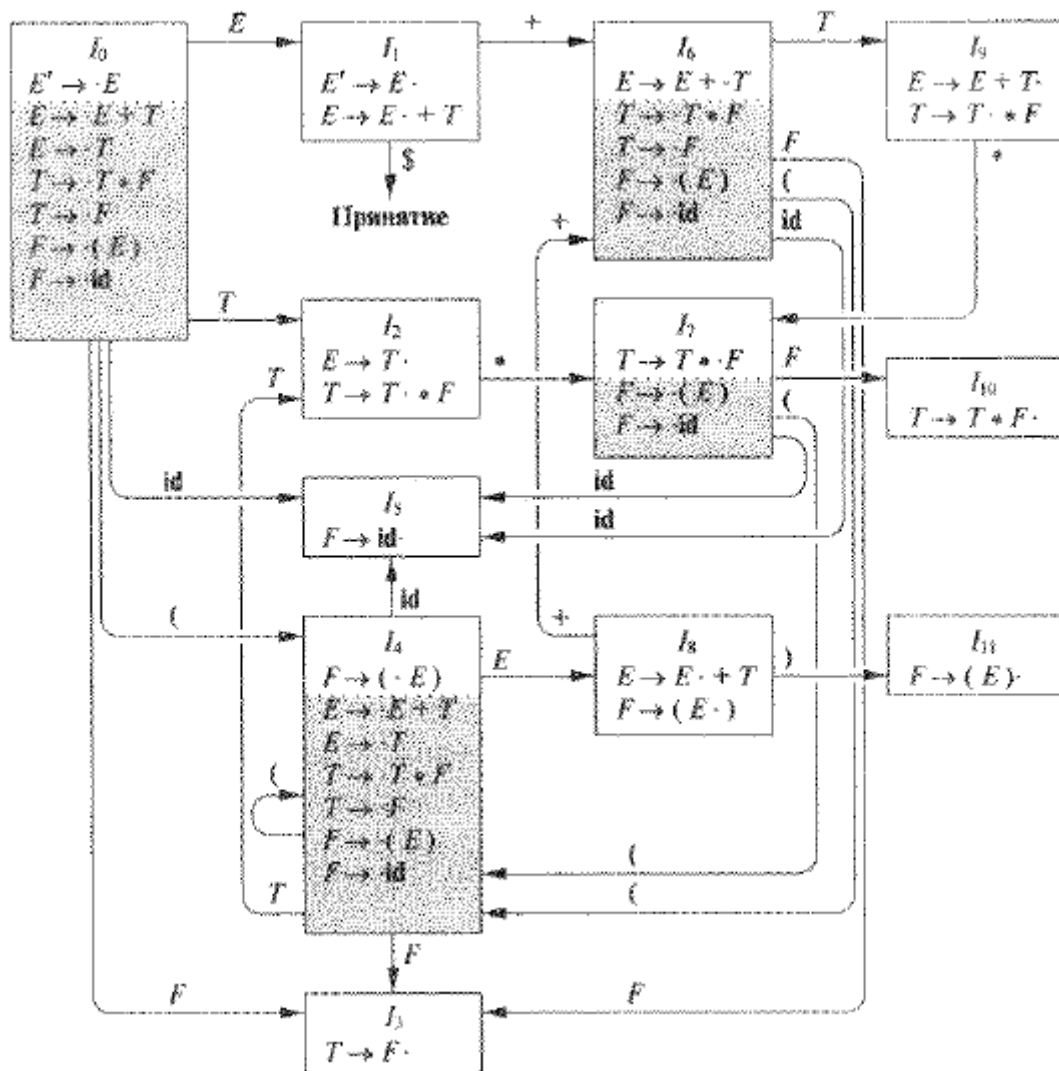


Рисунок 2 - канонический LR(0)-автомат для грамматики арифметических выражений

Задание функций CLOSURE и GOTO

Для построения канонического LR(0) – набора необходимо определить расширенную грамматику, функции CLOSURE и GOTO. Если G – грамматика со стартовым символом S , то расширенная грамматика G' представляет собой G с новым стартовым символом S' и продукцией $S' \rightarrow S$. Это нужно, чтобы показать синтаксическому анализатору, когда следует прекратить анализ и сообщить о принятии вхожей строки. Синтаксический разбор завершается, когда анализатор выполняет свертку с использованием продукции $S' \rightarrow S$.

Замыкание

Если I – множество пунктов грамматики G , то $CLOSURE(I)$ – множество пунктов построенное из I согласно двум правилам:

1. Замыкание содержит все пункты из I .
2. Если $A \rightarrow \alpha.B\beta$ входит в $CLOSURE(I)$, а $B \rightarrow \gamma$ является продукцией, то в $CLOSURE(I)$ добавляется пункт $B \rightarrow \gamma$, если его еще там нет. Правильно применяется, пока не останется пунктов, которые могут быть добавлены в замыкание.

Эти правила предполагают, что в процессе синтаксического анализа можно встретить подстроку, порождаемую $B\beta$. Она будет иметь префикс, порождаемый из B путем применения одного из правил B .

В свою очередь множество пунктов можно разделить на два класса:

1. Базисные пункты или пункты ядра (kernel): начальный пункт $S' \rightarrow .S$ и все пункты, у которых точки расположены не у левого края. Если одна B -продукция добавляется в замыкание I с точкой на левом конце, то в замыкание будут аналогичным образом добавлены все B -продукции. Поэтому при некоторых условиях нет необходимости в перечислении пунктов $B \rightarrow \gamma$, добавленных в I при помощи функции $CLOSURE$; Достаточно списка нетерминалов B , продукции которых были добавлены таким образом.
2. Небазисные пункты, у которых точки расположены слева, за исключением $SS' \rightarrow .S$.

Все небазисные пункты могут быть восстановлены замыканием, поэтому можно их отбросить и сэкономить память программы.

Функция GOTO

Функция $GOTO(I, X)$, где I -множество пунктов, а X -грамматический символ, определяется как замыкание множества всех пунктов $A \rightarrow \alpha X \beta$ таких, что $A \rightarrow \alpha \cdot X \beta$ находится в I .

Эта функция используется для переходов в LR(0)-автомате грамматики.

Построение LR(0)-автомата

Состояния автомата соответствуют множествам пунктов и $GOTO(I, X)$ указывает переход из состояния I при входном символе X .

Стартовое состояние LR(0)-автомата $CLOSURE(S' \rightarrow \cdot S)$.

Автомат должен принимать решения по текущему действию – переносу или свертке. Пусть Пусть строка γ из символов грамматики переводит LR(0)-автомат из состояния 0 в некоторое состояние j . Тогда выполняется перенос очередного входного символа a , если состояние j имеет переход для данного символа. Иначе выполняется свертка, пункт в состоянии j показывает, какую продукции следует использовать.

Алгоритм LR-анализа использует стек для отслеживания состояний. Символы грамматики могут быть восстановлены из состояний. Анализ состоит из входного буфера, выхода, стека, программы и таблицы синтаксического анализа, содержащей ACTION и GOTO. Программа синтаксического анализа по одному считывает символы из входного буфера. Пс-анализатор должен перенести символ, LR-анализатор переносит состояние.

Стек хранит последовательность состояний $s_0 s_1 \dots s_n$, которые находятся на вершине стека. В соответствии с построением каждое состояние имеет соответствующий символ грамматики. Состояния соответствуют множествам пунктов и существует переход из состояния i в состояние j , если

$GOTO(I_i, X) = I_j$. Все переходы в состояние j должны соответствовать одному и тому же символу грамматики X . Таким образом, каждое состояние, кроме 0, имеет единственный грамматический символ, связанный с ним. В

Таблицы ACTION и GOTO

Таблица синтаксического анализа состоит из двух частей: функции действий синтаксического анализа ACTION и функции переходов GOTO.

Функция ACTION принимает в качестве аргумента состояние i и терминал a (или $\$$ - маркер конца входной строки). Значение ACTION[i, a] имеет один из следующих видов:

- а) Перенос j , где j – состояние. Синтаксический анализатор переносит входной символ на стек, но для представления символа использует состояние j .
- б) Свертка $A \rightarrow \beta$. Действие синтаксического анализатора состоит в эффективной свертке β на вершине стека в заголовок A .
- в) Принятие. Синтаксический анализатор принимает входную строку и завершает анализ.
- г) Ошибка. Синтаксический анализатор обнаруживает ошибку во входной строке и предпринимает некоторое корректирующее действие.

Функция GOTO распространяется на состояние: если $GOTO(I_i, A) = I_j$, то GOTO отображает также состояния i и нетерминал A на состояние j .

Поведение LR-анализатора

Очередной шаг синтаксического анализатора можно определить, зная текущее состояние с вершины стека и текущий входной символ a_i , путем обращения к записи ACTION[s_m, a_i]. Возможны четыре ситуации:

a) $ACTION[s_m, a_i]$ = перенос s . Синтаксический анализатор переносит в стек очередное состояние s . Текущим входным символом становится a_{i+1} .

$ACTION[s_m, a_i]$ = свертка $A \rightarrow \beta$. Действие синтаксического анализатора состоит в эффективной свертке β на вершине стека в заголовок A . В процессе свертки учитывается длина правой части продукции свертки $\beta - r$, а состояние $s = s_{m-r}$. При свертке текущий входной символ не изменяется.

b) Принятие. Синтаксический анализатор принимает входную строку и завершает анализ.

c) Ошибка. Синтаксический анализатор обнаруживает ошибку во входной строке и предпринимает некоторое корректирующее действие, например, вызывает подпрограмму восстановления после ошибки.

Приведенных выше инструментов достаточно, чтобы построить SLR-анализатор (Simple LR).

LALR-анализатор

Существуют более мощные LR-анализаторы: канонический LR и LR с предпросмотр или LALR. LALR основан на множестве пунктов LR(0) и имеет меньше состояний, чем типичный анализатор, основанный на LR(1)-пунктах. Этот метод позволяет работать с существенно большим количеством грамматик за счет добавления предпросмотров в LR(0)-пункты, при этом таблицы не больше, чем в SLR.

Пункты и переопределение процедур

LR(1)-пунктом называется объект $[A \rightarrow \alpha.\beta, a]$, где $A \rightarrow \alpha\beta$ -продукция, а a -терминал или маркер конца входной строки $\$$. 1 обозначает длину предпросмотра.

Метод построения набора множеств допустимых LR(1)-пунктов отличается только модификацией процедур CLOSURE и GOTO.

Для определения новой процедуры CLOSURE рассмотрим пункт вида $[A \rightarrow \alpha.B\beta, a]$ в множестве пунктов, допустимых для некоторого активного префикса γ . Тогда существует правое порождение $S \Rightarrow_{rm}^* \delta a B \beta a$, где $\gamma = \delta \alpha$. Предположим, что $\beta a x$ порождает строку терминалов $b y$. Тогда для каждой продукции вида $B \rightarrow \eta$ для некоторого η имеется порождение $S \Rightarrow_{rm}^* \gamma B b y \Rightarrow_{rm} \gamma \eta b y$. Таким образом, $[B \rightarrow \cdot \eta, b]$ является допустимым для γ . Заметим, что b может быть первым терминалом, порожденным из β , либо возможно, что β порождает ε в порождении $\beta a x \Rightarrow_{rm}^* b y$, и, следовательно, b может представлять собой a . Поэтому b может быть любым терминалом в $FIRST(\beta a x)$. При этом x не может содержать первый терминал из $b y$, так что $FIRST(\beta a x) = FIRST(\beta a)$.

Канонические таблицы LR(1)-анализа

Приведем построение LR(1)-функций ACTION и GOTO из множеств LR(1)-пунктов. Отличие заключается в значениях записей таблицы.

Построение происходит по следующему алгоритму:

1. Построить $C' = \{I_0, I_1, \dots, I_n\}$ — набор множеств LR(1)-пунктов для G' .
2. Состояния синтаксического анализатора строятся из I_i . Действие синтаксического анализа для состояния i определяется следующим образом:
 1. Если $[A \rightarrow \alpha.a\beta, b]$ входит в I_i и $GOTO(I_i, a) = I_j$, установить ACTION[i, a] равным «перенос j». Здесь a должно быть терминалом.
 2. Если $[A \rightarrow \alpha., a]$ входит в I_i и $A \neq S'$, установить ACTION[i, a] равным «свертка $A \rightarrow \alpha$ ».
 3. Если $[S' \rightarrow S., \$]$ входит в I_i , установить ACTION[i, \$] равным «принятие».

4. При применении указанных правил обнаруживаются конфликтующие действия, грамматика не является LR(1).
3. Переходы для состояния i строятся для всех нетерминалов A с использованием правила: если $GOTO(I_i, A) = I_j$, то $GOTO(i, A) = j$.
4. Все записи, не определенные правилами 2 и 3, считаются ошибкой.
5. Начальное состояние синтаксического анализатора – состояние, построенное из множества пунктов, содержащего $[S' \rightarrow S., \$]$.

Таблица, построенная таким образом, называется канонической таблицей LR(1)-анализа.

Построение LALR-таблиц синтаксического анализа

В этом методе таблицы получаются сравнительно небольшими по сравнению с каноническими LR-таблицами и большинство синтаксических конструкций языков программирования могут быть легко выражены LALR-грамматикой.

При этом SLR- и LALR-таблицы для грамматики почти всегда имеют одно и то же количество состояний.

Можно рассмотреть множество LR(1)-пунктов, имеющих одно и то же ядро, т.е. множество первых компонентов, и объединить эти множества с общими ядрами в одно множество пунктов.

Поскольку ядро множества $GOTO(I, X)$ зависит только от ядра множества I , значения функции $GOTO$ объединяемых множеств также могут быть объединены. Функция ACTION должна быть изменена, чтобы отражать не ошибочные действия всех объединяемых множеств пунктов.

Пусть имеется LR(1)-грамматика, т.е. ее множество пунктов не вызывает конфликтов действий. Можно заменить все состояния, имеющие одно и то же ядро, их объединениями. Если предположить, что в объединении возникает

конфликт при просмотре входного символа a , поскольку существует пункт $[A \rightarrow \alpha.a]$, вызывающий свертку по продукции $\rightarrow \alpha$, а также другой пункт, $[B \rightarrow \beta.a\gamma, b]$, приводящий к переносу. Тогда некоторое множество пунктов, из которого было сформировано объединение, имело пункт $[A \rightarrow \alpha.a]$. Поскольку ядра объединяемых множеств совпадают, это множество должно также иметь пункт $B \rightarrow \beta.a\gamma, c]$ для некоторого c . Но в таком случае это состояние имело бы конфликт переноса/свертки для символа a и вопреки предположению грамматика бы не была LR(1)-грамматикой. Следовательно, объединение состояний с одинаковыми ядрами не может привести к конфликту переноса/свертки, если такой конфликт не присутствовал ни в одном из исходных состояний (поскольку переносы зависят только от ядра, но не от предпросмотра).

Все же возможно появление конфликта свертка/свертка. Такой пример рассматривается в книге [1], пункт 4.42.

Начальный вариант построения LALR-таблицы

1. Строится набор множеств LR(1)-пунктов $C = \{I_0, I_1, \dots, I_n\}$ для расширенной грамматики.
2. Для каждого ядра, имеющего среди множества LR(1)-пунктов, находим все множества, имеющие это ядро, и заменяем эти множества их объединением.
3. Пусть $C = \{J_0, J_1, \dots, J_m\}$ – полученные в результате множества LR(1)-пунктов. Функция ACTION для состояние i строится из J_i так же, как в алгоритме построения канонической таблицы LR(1)-анализа. Если обнаруживается конфликт, то грамматика не является LALR(1)-грамматикой.
4. Строится таблица GOTO. Если J – объединение одного или нескольких множеств LR(1)-пунктов, т.е. $J = I_0 \cup I_1 \cup \dots \cup I_k$, то ядра множеств $GOTO(I_1, X), GOTO(I_1, X) \dots GOTO(I_k, X)$ одни и те же, поскольку

I_1, I_2, \dots, I_k имеют одно и то же ядро. Пусть K – объединение всех множеств пунктов, имеющих одно и то же ядро. Обозначим через K объединение всех множеств пунктом, имеющих одно и то же ядро, что и $GOTO(I_1, X)$. Тогда $GOTO(J, X) = K$.

Полученная таблица называется таблицей LALR-анализа. Если конфликты отсутствуют, то грамматика называется LALR(1).

Эффективное построение LALR(1)-таблицы

Если внести некоторые изменения, то можно избежать построения набора множеств LR(1)-пунктов в процессе создания таблицы LALR(1)-анализа.

- 1) Можно построить ядра LALR(1)-пунктов на основе ядер LR(0)-пунктов при помощи процесса распространения и спонтанной генерации символов предпросмотра.
- 2) Если имеются ядра LALR(1), то можно сгенерировать LALR(1)-таблицу путем замыкания каждого ядра с использованием функции CLOSURE, а затем вычислить запись таблицы аналогично алгоритму построения канонической таблицы LR(1)-анализа, как если бы LALR(1)-пунктов были каноническими LR(1) множествами пунктов.

Далее для создания ядер множеств LALR(1)-пунктов требуется назначить корректные предпросматриваемые символы LR(0)-пунктам ядер. Существует два способа, которыми предпросматриваемый символ b может быть назначен LR(0)-пункту $B \rightarrow \gamma. \delta$ из некоторого множества LALR(1)-пунктов J .

Существует множество пунктов I с базисным пунктом $A \rightarrow \alpha. \beta, a$, $J = GOTO(I, X)$ и построение

$$GOTO(CLOSURE(\{[A \rightarrow \alpha. \beta, a]\}), X)$$

содержит $[B \rightarrow \gamma. \delta, b]$ только потому, что одним из связанных с $[A \rightarrow \alpha. \beta]$ предпросматриваемых символов является b . В этом случае имеет место быть

распространение символа предпросмотра от $A \rightarrow \alpha.\beta$ в ядре I к $B \rightarrow \gamma.\delta$ в ядре J . При этом распространение не зависит от конкретного символа предпросмотра; либо все символы предпросмотра распространяются от одного пункта к другому, либо ни один из них.

Необходимо определить спонтанно генерируемые символы предпросмотра для каждого множества LR(0)-пунктов, а также определить, для каких пунктов происходит распространение предпросмотров и из каких именно пунктов.

Пусть $\#$ - символ, отсутствующий в заданной грамматике и $A \rightarrow \alpha.\beta$ - ядро LR(0)-пункта в множестве I . Для каждого X вычислим $J = GOTO(CLOSURE(\{[A \rightarrow \alpha.\beta, \#]\}, X)$ Для каждого базисного пункта в J нужно проверить его множество символов предпросмотра. Если $\#$ является символом предпросмотра, то предпросмотры распространяются к этому пункту от $A \rightarrow \alpha.\beta$. Все прочие предпросмотры генерируются спонтанно.

Алгоритм определения предпросмотров

На вход подается ядро K множества LR(0)-пунктов I и символ грамматики X .

На выходе программа должна передавать спонтанно генерируемые пунктами из I , от которых предпросмотры распространяются к базисным пунктам в $GOTO(I, X)$.

Для каждого пункта продукции $A \rightarrow \alpha.\beta$ из K определяется замыкание $J = CLOSURE(\{[A \rightarrow \alpha.\beta, \#]\})$. Если продукция $[B \rightarrow \gamma.X\delta, a]$ принадлежит замыканию и текущий символ не является символом конца разбора, то символ предпросмотра a спонтанно генерируется для пункта $B \rightarrow \gamma.X.\delta$ в $GOTO(I, X)$. Иначе если продукция $[B \rightarrow \gamma.X\delta, \#]$ принадлежит замыканию, это значит, что символы предпросмотра распространяются от $A \rightarrow \alpha.\beta$ из I к $B \rightarrow \gamma.X.\delta$ в $GOTO(I, X)$.

Для формирования множеств LALR(1)-пунктов нужно назначить предпросмотры ядрам множеств LR(0)-пунктов. Символ \$ является символом предпросмотра для $S' \rightarrow .S$ в исходном множестве LR(0)-пунктов.

Алгоритм, приведенный выше, дает спонтанно генерируемые символы предпросмотра. После вычисления они распространяются до тех пор, пока не останется ни одного возможного распространения.

Вычисление ядер наборов множеств LALR(1)-пунктов

Алгоритм по заданной расширенной грамматике вычисляет ядра наборов множеств LALR(1)-пунктов. Для этого нужно выполнить следующие действия:

1. Построить ядра множеств LR(0)-пунктов грамматики. В зависимости от требований к используемому объему памяти это можно сделать либо по алгоритму, который строит LR(0)-автомат, либо можно сохранять только базисные пункты и вычислять GOTO для множества пунктов I, сначала вычисляя замыкание I.
2. Применение алгоритма определения предпросмотров к ядру каждого множества LR(0)-пунктов и грамматическому символу X, чтобы определить, какие символы предпросмотра спонтанно генерируются для базисных пунктов в $GOTO(I, X)$ и из каких пунктов I символов предпросмотра распространяются на базисные пункты $GOTO(I, X)$.
3. Инициализируется таблица, которая для каждого базисного пункта в каждом множестве пунктов дает связанные с ними предпросмотры. Изначально с каждым пунктом связаны только те, предпросмотры, которые в п.2 определены как сгенерированные спонтанно.
4. Повторяются проходы по базисным пунктам во всех множествах. При посещении пункта i с помощью таблицы, построенной в пункте 2, ищутся базисные пункты, на которые i распространяет свои

предпросмотры. Текущее множество предпросмотров для i добавляется к множествам, связанным с каждым из пунктов, на которые i распространяет свои предпросмотры. Такие проходы по базисным пунктам выполняются до тех пор, пока не останется новых предпросмотров для распространения.

РЕАЛИЗАЦИЯ

Входные данные

На вход программы подается строка и задаются правила грамматики. Рассмотрим каждый пункт по отдельности.

Терминалы

Терминалы в данной программе задаются как показано в Листинге 1.

Первым аргументом конструктора является регулярное выражение, по которому можно распознать данный терминал, а вторым аргументов функция, которую следует применить на данный терминал. Например, цифра задается регулярным выражением '[0-9]+' и функцией, которая будет преобразовывать строку в целочисленный тип.

```
digit = pe.Terminal('[0-9]+', int)
D |= digit
```

Листинг 1 — явное задание терминала и его использование в правиле грамматики

Терминалы могут также задаваться неявно, когда появляются в правой части продукции. Пример приведен в Листинге 2. Здесь «+» является терминальным символом, что устанавливается непосредственно в процессе разбора грамматики.

```
E |= T, '+', T, lambda t1, t2: t1 + t2
```

Листинг 2 — неявное задание терминала в правой части продукции

Нетерминальные символы и правила грамматики

Стартовый символ грамматики можно задать явно, иначе стартовым символом является нетерминал первого правила грамматики.

Для указания символа как нетерминального используется конструкция, указанная в Листинге 3.

Листинг 3 — объявление нетерминального символа

Вызывается конструктор класса `NonTerminal`, на вход подается только имя нетерминала.

Задание правила представлено на Листинге 2.

Оператор «`|=`» указывает на то, что передается правило. Для этого символ перегружается. В левой части выражения указывается нетерминал, в правой само правило. В данном примере задается правило, где `T` — другой нетерминальный символ. Здесь в левой части правила указывается нетерминальный символ, `'+'` является терминальным символом, а затем передается лямбда-функция, которая указывает на выполняемое действие при свертке по этому правилу. Если лямбда-функция не указана, то в процессе создания грамматики в нетерминал добавляется функция по умолчанию: $\lambda id \rightarrow id$, т.е. функция не выполняет никаких преобразований.

Важно отметить, что символы, заданные непосредственно в правиле имеют приоритет перед терминальными символами, заданными явно через класс терминалов. Так, если существует регулярное выражение, распознающее плюс, то этот символ все равно будет восприниматься так, как это задано в правой части правила.

Эти конструкции полностью задают грамматику `G`, представленную в Листинге 4.

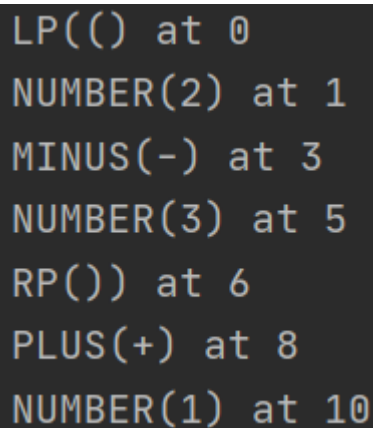
```
E = pe.NonTerminal('E')
T = pe.NonTerminal('T')
F = pe.NonTerminal('F')
N = pe.NonTerminal('N')
D = pe.NonTerminal('D')
digit = pe.Terminal('[0-9]+', int)
E |= T, '+', T, lambda t1, t2: t1 + t2
E |= T
E |= T, '-', T, lambda t1, t2: t1 - t2
T |= F, '*', F, lambda f1, f2: f1 * f2
T |= F, '/', F, lambda f1, f2: f1 / f2
T |= F
F |= N
F |= '(', E, ')'
N |= D
N |= D, D
D |= digit
```

Листинг 4 — задание грамматики

Лексический анализ

Входная строка подается явно в виде строки, либо читается из файла.

Лексический анализатор получает на вход строку и возвращает список токенов. Пусть входная строка представляет простое арифметическое выражение «(2 – 3) + 1». Тогда лексический анализатор вернет список токенов, представленный на рисунке 3.



```
LP(( ) at 0
NUMBER(2) at 1
MINUS(-) at 3
NUMBER(3) at 5
RP( ) at 6
PLUS(+) at 8
NUMBER(1) at 10
```

Рисунок 3 - анализатор вернул список токенов по входной строке

Токены могут относиться либо к классу Token, либо к классу AttrToken – то есть у токена есть атрибут, который необходимо будет положить на стек. На Листинге 5 поясняется разница в задании токена и токена с атрибутом:

```
three = AttrToken(digit, 3)
plus = Token('+')
```

Листинг 5 — пример объявления токена и токена с атрибутом

Лексический анализатор разбирает строку, последовательно применяя метод `nextToken()` к каждому символу. При создании грамматики, синтаксический анализатор распознает терминалы, заданные неявно в самом правиле, и добавляет их в список терминалов грамматики. В методе `nextToken()` происходит сопоставление символов с неявными терминалами и если терминал не найден, то происходит сопоставление с подходящим регулярным выражением. В случае, если символы подходят под разные регулярные выражения, то происходит сопоставление с наибольшим вхождением.

Синтаксический анализ

В начале синтаксического анализа по заданной грамматике G создается расширенная граматика G' за счет добавления правила $S' \rightarrow S$

Построение LALR(1)-таблицы

Процесс построения таблицы уже описан в теоретической части. В результате строится таблица, содержащая ACTION и GOTO для всех состояний и символов. Поскольку полученная таблица является довольно большой, то приводится только фрагмент таблицы.


```

State 1
  10  N: D .
  11  N: D . D

  for terminal -: reduce using rule 10
  for terminal ): reduce using rule 10
  for terminal 2: shift and go to state 6
  for terminal /: reduce using rule 10
  for terminal *: reduce using rule 10
  for terminal +: reduce using rule 10
  for terminal 1: shift and go to state 7
  for terminal 0: shift and go to state 9
  for terminal <parser_edsl.Terminal object at 0x016545F8>: shift and go to state 10
  for terminal $end: reduce using rule 10

  for non-terminal D: go to state 11

State 2
  0    $accept: E .

```

Рисунок 4 - фрагмент LALR-таблицы

Парсер

Парсер получает на вход список токенов и для каждого токена определяет следующее действие: перенос, свертка, ошибка или принятие. Парсер использует два стека:

- 1) Действие «переноса». На стек атрибутов кладется атрибут, соответствующий лексеме, либо не кладется ничего в случае, если у токена атрибутов нет. На второй стек кладется номер состояния, соответствующий таблице ACTION.
- 2) Действие «свертки». В случае свертки на стек так же кладется атрибут. Для стека состояний вычисляется число аргументов лямбда-функции, т.е. число аргументов продукции, согласованное с ACTION. Затем из стека состояний удаляется соответствующее число аргументов и добавляется новое состояние, взятое из таблицы GOTO.
- 3) «Ошибка». Анализатор выдает ошибку.

4) «Принятие». Разбор оканчивается состоянием «принятие», т.е. он завершился успешно.

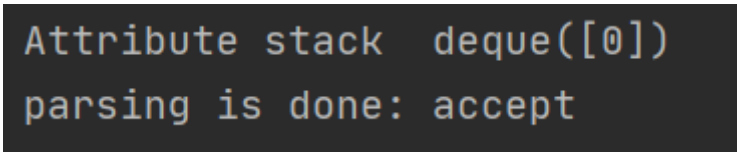
ТЕСТИРОВАНИЕ

Для тестирования была взята грамматика, содержащая выражения с арифметическими операциями «+», «-», «*», «/» и скобками. Терминальные символы задаются как явные экземпляры класса и неявно при задании продукции. Продукции могут содержать заданные пользователем лямбда-функции. В случае, если функция не задана, то она по умолчанию соответствует функции тождества. Исходная грамматика представлена в Листинге 4.

После задания грамматики выполняется построение LALR(1)-таблицы, содержащей состояния и таблицы ACTION и GOTO. Фрагмент таблицы представлен на Рисунке 4.

На вход, например, подается строка « $(2 - 3) + 1$ ». В результате работы программы лексический анализатор возвращает список токенов (Рисунок 3).

Затем проводится синтаксический анализ, парсер выбирает одно из четырех возможных действий и при необходимости изменяет стек состояний и атрибутов. В результате выполнения программы достигается конечное состояние - «принятие» и на стеке атрибутов остается вычисленное значение - 0.



```
Attribute stack deque([0])
parsing is done: accept
```

Рисунок 5 - результат работы парсера

ЗАКЛЮЧЕНИЕ

В результате выполнения работы была получена программа, способная по заданной грамматике построить LALR(1)-таблицу. Также изначально грамматика проверяется на принадлежность к классу LR(1) грамматик.

По заданной исходной строке происходит лексический анализ. Полученные токены затем подаются на вход парсеру, который в соответствии с полученной таблицей выполняет синтаксический анализ типа «перенос-свертка». провести разбор конкретной строки, которая вначале обрабатывается лексическим анализатором.

Таким образом, пользователь получает удобную библиотеку лексического и синтаксического анализа по любой заданной LALR(1)-грамматике.

Работу можно расширить добавлением в программу построения дерева разбора. Затем его можно отрисовать, например, с помощью программы GRAPHVIZ. Тогда пользователь сможет наглядно понять процесс синтаксического анализа.

СПИСОК ЛИТЕРАТУРЫ

1. Компиляторы. Принципы, технологии и инструментарий / А. Ахо, Р. Сети, М Лам, Д. Ульман. – Диалектика, 2019. – 1184 с. – ISBN 978-5-8459-1932-8
2. Коновалов А.В. – Конспект лекций по курсу «Конструирование компиляторов».
3. Непейвода А.Н. – Конспект лекций по курсу «Теория формальных языков».