Лекция 11. Синтаксический анализ. Часть 3

11.1 Детерминированный разбор с помощью алгоритма «перенос-свертка».....1
11.2 LR(0)-автоматы.....5
11.3 Алгоритм LR-анализа.....8
11.4 Построение таблиц SLR-анализа.....10
11.5 Активные префиксы.....12
Литература к лекции 11.....13

Главные вопросы, которые мы обсуждаем, представлены на СЛАЙДЕ 1. Мы рассматриваем широкий класс КСГ, для которых возможен детерминированный ВСА. Данный класс – LR(*k*)-грамматики нашел широкое применение при проектировании компиляторов.

11.1 Детерминированный разбор с помощью алгоритма «перенос-свертка»

Анализ входной строки таким алгоритмом состоит в переносе входных символов в магазин до тех пор, пока в его вершине не встретится основа. В этот момент производится **свертка** (*reduction*), в результате которой основа заменяется на LHS продукции с основой. Этот процесс продолжается до тех пор, пока не будет считана вся входная строка, а в магазине не останется аксиома, или алгоритм не выдаст сообщение об ошибке.

Ключевые решения, принимаемые в процессе ВСА – когда выполнять свертку, и какую продукцию применить.

Пример 71.

Пусть задана КСГ, показанная на СЛАЙДЕ 2. Там же показана последовательность сверток для анализируемой строки *id***id*.

Строки этой последовательности образованы корнями всех поддеревьев на схеме. Последовательность начинается с входной строки. Первая свертка дает *F***id* путем свертки крайнего слева *id* в *F* с использованием продукции *F* -> *id*. Вторая свертка дает *T***id* при помощи свертки *F* в *T*. После этого у нас есть выбор между сверткой строки *T*, которая является RHS продукции *E* -> *T*, и строки, состоящей из следующего *id*, являющегося RHS продукции *F* -> *id*. Вместо того, чтобы выполнять свертку *T* в *E*, свернем второй *id* в *F*, получая строку *T***F*. Затем эта строка сворачивается в *T*. Синтаксический анализ завершается сверткой *T* в аксиому E.

По определению свертка представляет собой шаг, обратный порождению (в порождении нетерминал в СФ замещается RHS одной из его продукций). Цель ВСА, таким образом, состоит в построении **обратного порождения**. Наш случай:

$E \Rightarrow_r T \Rightarrow_r T*F \Rightarrow_r T*id \Rightarrow_r F*id \Rightarrow_r id*id$.

Данное порождение является **правым**.

ВСА в процессе чтения строки слева направо строит обращенное правое порождения. Напомним неформально, что **основа** строки – это подстрока, которая соответствует телу продукции и свертка которой представляет собой один шаг обращенного правого порождения.

Например, основы в процессе синтаксического анализа $id_1 * id_2$ (нижние индексы добавлены к *id* для ясности) в соответствии с нашей КСГ показаны на СЛАЙДЕ 3.

Хотя *T* – RHS продукции *E* -> *T*, символ *T* не является основой в СФ $T * id_2$. Если заменить *T* на *E*, мы получим строку $E * id_2$, которая не может быть порождена из *E*. Таким образом, крайняя слева подстрока, которая соответствует RHS некоторой продукции, не обязательно является основой.

Формально, если $S \Rightarrow_r^* \alpha A w \Rightarrow_r \alpha\beta w$, как на СЛАЙДЕ 3, то продукция A -> β в

позиции после α является **основой** (*handle*) $\alpha\beta w$. В качестве альтернативы основой правосентенциальной формы γ является продукция A -> β в позиции γ, в которой может быть найдена строка β, такая, что замена β в данной позиции на A дает предшествующую правосентенциальную форму в правом порождении γ.

Заметим, что строка w справа от основы должна содержать только терминальные символы. Для удобства мы будем говорить о теле продукции β как об основе, а не обо всей продукции A -> β в целом. Следует также заметить, что грамматика может быть неоднозначной, с несколькими правыми порождениями $\alpha\beta w$. Если грамматика однозначна, то каждая правосентенциальная форма грамматики имеет ровно одну основу.

Обращенное правое порождение может быть получено посредством «обрезки основ» (СЛАЙД 4). Мы начинаем процесс со строки терминалов, которую хотим проанализировать. Если w – строка рассматриваемой грамматики, то пусть $w = \gamma_n$, где γ_n – n-я правосентенциальная форма некоторого еще неизвестного правого порождения $S \Rightarrow_r \gamma_0 \Rightarrow_r \gamma_1 \Rightarrow_r \gamma_2 \Rightarrow_r^* \ldots \Rightarrow_r \gamma_{n-1} \Rightarrow_r \gamma_n \Rightarrow_r w$. Для воссоздания этого обратного порождения в обратном порядке мы находим основу β_n в γ_n и заменяем ее LHS продукции A_n -> β_n для получения предыдущей правосентенциальной формы γ_{n-1}. Пока мы не знаем, каким образом искать основы, но вскоре познакомимся с соответствующими методами.

Затем мы повторяем описанный процесс, т.е. находим в γ_{n-1} основу β_{n-1} и свертываем ее для получения правосентенциальной формы γ_{n-2}. Если после очередного шага правосентенциальная форма содержит только аксиому S, то мы прекращаем процесс и сообщаем об успешном завершении анализа. Обращенная последовательность продукций, использованных в свертках, представляет собой правое порождение входной строки.

Синтаксический анализ «Перенос-свертка»

СА «Перенос-свертка» (далее ПС-анализ) представляет собой разновидность ВСА, в которой для хранения символов грамматики используется магазин (стек), а для хранения остающейся непроанализированной части входной строки – входной буфер.

Мы используем символ $ для маркирования дна стека и правого конца входной строки. При рассмотрении ВСА удобно располагать вершину магазина (стека). Изначально стек пуст, а во входном буфере находится строка w:

Стек	Вход
$	w

В процессе чтения входной строки слева направо синтаксический анализатор выполняет нуль или несколько переносов символов в стек, пока не будет готов выполнить свертку строки β символов грамматики на вершине стека. Затем он выполняет свертку β к LHS соответствующей продукции. Синтаксический анализатор повторяет этот цикл до тех пор, пока не будет обнаружена ошибка или пока стек не будет содержать только аксиому, а входной буфер будет при этом пуст:

Стек	Вход
$$S$	$

Достигнув указанной конфигурации, синтаксический анализатор останавливается и сообщает об успешном завершении анализа. На СЛАЙДЕ 5 пошагово показаны действия ПС-анализатора, выполняемые при синтаксическом анализе строки id_1 * id_2 согласно грамматике выражений.

Хотя основными операциями являются перенос и свертка, всего ПС-анализатор может выполнять четыре действия: 1) перенос, 2) свертка, 3) принятие и 4) ошибка. Рассмотрим их чуть подробнее (СЛАЙД 5).

1. Под **переносом** (*shift*) понимается перемещение очередного входного символа на вершину стека.

2. Под **сверткой** (*reduce*) понимается следующая операция. На вершине стека должна располагаться RHS сворачиваемой строки. Определяется левый конец строки в стеке и принимается решение о том, каким нетерминалом будет заменена строка.

3. Под **принятием** (*accept*) понимается объявление об успешном завершении СА.

4. Обнаружение синтаксической **ошибки** (*error*) и вероятный вызов подпрограммы восстановления после ошибки.

Использование стека в ПС-анализаторе объясняется тем важным фактом, что основа всегда находится на вершине стека и никогда – внутри него. Это можно показать путем рассмотрения возможных видов двух последовательных шагов в любом правом порождении. На СЛАЙДЕ 6 показаны эти два возможных случая. В случае (1) A заменяется на βBy, после чего крайний справа нетерминал B в βBy заменяется на γ. В случае (2) A снова раскрывается первым, но на этот раз RHS представляет собой строку y, состоящую из одних терминалов. Следующий крайний справа нетерминал B будет находиться где-то слева от y.

Иначе говоря, имеем следующее:

1. $S \Rightarrow_r \alpha Az \Rightarrow_r \alpha\beta Byz \Rightarrow_r \alpha\beta\gamma yz$.

2. $S \Rightarrow_r \alpha BxAz \Rightarrow_r \alpha Bxyz \Rightarrow_r \alpha\gamma xyz$.

Первый случай можно рассмотреть в обратном направлении с момента попадания анализатора в конфигурацию

Стек	Вход
$\$\alpha\beta\gamma$	$yz\$$

Основа γ сворачивается в B, анализатор переходит в другую конфигурацию.

Стек	Вход
$\$\alpha\beta B$	$yz\$$

Теперь синтаксический анализатор может выполнить перенос строки y в стек при помощи нуля или нескольких шагов переноса и получить конфигурацию

Стек	Вход
$\$\alpha\beta By$	$z\$$

На вершине стека основа βBy, можно выполнить свертку в A.

Аналогично поступим во втором случае. В конфигурации

Стек	Вход
$\$\alpha\gamma$	$xyz\$$

На вершине стека находится основа γ, ее можно свернуть в B. После этого анализатор может перенести xy для получения очередной основы y на вершине стека. Ее тоже можно свернуть к A.

Стек	Вход
$\$\alpha Bxy$	$z\$$

В обоих случаях анализатор после свертки должен выполнить один или более раз перенос символов для получения в стеке очередной основы. Он никогда не должен углубляться в стек в ее поисках.

Конфликты в процессе ПС-анализа

Имеются КСГ, для которых ПС-анализ неприменим. Любой ПС-анализатор для такой грамматики может достичь конфигурации, в которой синтаксический анализатор, обладая информацией о содержимом стека и очередных k входных символах, не может принять решение о том, следует ли выполнить перенос или свертку (это **конфликт «перенос/свертка»**) либо какое именно из нескольких сверток должно быть выполнено (**конфликт «свертка/свертка»**). СЛАЙД 7.

Ниже мы рассмотрим несколько примеров синтаксических конструкций, которые приводят к таким грамматикам. Технически эти грамматики не принадлежат классу LR(k)-грамматик, определенному далее. Будем говорить о них как о не-LR-грамматиках. Символ k в LR(k) указывает количество символов, которые «предпросматриваются» во входном потоке. Обычно используемые грамматики принадлежат классу LR(1), т.е. выполняется предпросмотр не более одного символа.

Пример 72.

Пусть задана КСГ, показанная на СЛАЙДЕ 8. Если анализатор находится в конфигурации

Стек	Вход
...**if** *expr* **then** *stmt*	*else*…$

то анализатор не может определиться, является ли ***if*** *expr* ***then*** *stmt* основой без использования символов ниже ее в стеке. Здесь имеет место конфликт «перенос/свертка». В зависимости от того, что следует за ***else*** во входном потоке, верным решением может оказаться свертка ***if*** *expr* ***then*** *stmt* в *stmt* или перенос *else* и поиск еще одного *stmt* для завершения альтернативы ***if*** *expr* ***then*** *stmt* ***else*** *stmt*.

При всем при том, ПС-анализ может быть адаптирован к разбору некоторых неоднозначных грамматик, таких, как приведенная выше. При разрешении указанного конфликта «перенос/свертка» при обнаружении ***else*** в пользу переноса синтаксический анализатор будет работать так, как мы от него и ожидаем, связывая ***else*** с предыдущим ***then***, которому еще не найдено соответствующее ***else***.

Еще одна распространенная причина конфликта – когда у нас есть основа, но содержимого стека и очередного входного символа недостаточно для определения продукции, которая должна использоваться в свертке. Следующий пример иллюстрирует эту ситуацию.

Пример 73.

Предположим, у нас есть анализатор, который возвращает имя *id* для всех имен независимо от их типа. Предположим также, что наш язык вызывает процедуры по именам с параметрами, заключенными в скобки, и что тот же синтаксис используется и для работы с массивами. Поскольку трансляции индексов массива и параметров процедуры существенно отличаются друг от друга, мы должны использовать различные продукции для порождения списка фактических параметров и индексов. Следовательно, наша грамматика может иметь (среди прочих) продукции, наподобие приведенных на СЛАЙДЕ 9.

Инструкция, начинающаяся с *p*(*i*, *j*), для синтаксического анализатора будет выглядеть как символы ***id*** (***id***, ***id***). После переноса первых трех символов в стек ПС-анализатор окажется в конфигурации:

Стек	Вход
...**id** (**id**	, ***id***)…

Очевидно, что *id* на вершине стека должен быть свернут, но какую продукцию использовать? Правильный выбор – пятая продукция, если *p* – имя процедуры или функции, и седьмая продукция, если *p* – массив. Содержимое стека не может подсказать, чем является *p*; для принятия решения мы должны использовать дополнительную информацию (если мы пишем транслятор, то она будет взята из таблицы символов, которая была занесена в стек при объявлении *p*).

Одно из решений проблемы заключается в замене ***id*** в первой продукции на *procid* и использовании более интеллектуального способа определения того, что собой представляет имя процедуры.

Если мы учтем это, то при обработке *p*(*i*, *j*) синтаксический анализатор может оказаться в конфигурации

Стек	Вход
... **procid** (**id**	, ***id***)…

Он также может оказаться в конфигурации, приведенной ранее. В первом случае мы выбираем свертку с использованием пятой продукции, в последнем – с использованием седьмой продукции. Сам выбор определяется третьим от вершины символом в стеке,

который даже не участвует в свертке. Для управления разбором ПС-анализатор может использовать информацию «из глубин» стека.

11.2 LR(0)-автоматы

Наиболее распространенный на сегодня тип восходящих синтаксических анализаторов основан на концепции, называющейся **LR(*k*)-анализом**. *L* здесь означает просмотр входной строки слева направо, *R* – построение обратного правого порождения, а *k* – количество предпросматриваемых символов входного потока, необходимое для принятия решения. Практический интерес представляют случаи $k = 0$ и $k = 1$. Мы рассмотрим только LR-анализаторы с $k \leq 1$. Если *k* не указано, то полагаем, что $k = 1$.

В этом и следующем разделах рассматриваются базовые концепции LR-анализа и простейший метод построения ПС-анализаторов, называющийся **простым** LR (*simple* LR, SLR). Определенное знакомство с базовыми концепциями весьма полезно даже в том случае, когда для построения LR-анализатора используется автоматический генератор анализаторов. Мы начнем с «ситуаций» и «состояний анализатора»; диагностические сообщения генератора LR-анализаторов обычно включают состояния синтаксического анализатора, которые могут использоваться для выяснения источника конфликтов синтаксического анализа.

LR-анализаторы управляются ТСА, как и LL-анализаторы. Грамматика, для которой можно построить ТСА с использованием метода, обсуждаемого ниже (и их аналогов), называется **LR-грамматикой**. Неформально, чтобы КСГ была LR, достаточно, чтобы синтаксический анализатор, работающий слева направо методом переноса-свертки, был способен распознавать основы правых СФ при их появлении на вершине стека.

Причины популярности LR-анализа (СЛАЙД 10):

1. Можно создавать анализаторы для распознавания основных конструкций языков программирования.

2. Это наиболее общий метод ПС-анализа без возврата, который не уступает другим, более примитивным методам.

3. LR-анализатор может обнаруживать ошибки на первом же некорректном входном символе.

4. Класс LR-грамматик представляет собой истинное надмножество грамматик, анализируемых LL-методом. Для LR(*k*)-грамматик мы должны быть способны распознать RHS в порожденной ею правосентенциальной форме с дополнительным предпросмотром *k* входных символов. Это требование существенно мягче требования для LL(*k*)-грамматик, в которых мы должны быть способны распознать продукцию по первым *k* символам порождения ее RHS. Таким образом, LR-грамматики могут описать существенно больше языков, чем LL-грамматики.

Основной недостаток LR-метода состоит в том, что построение LR-анализатора для грамматики типичного ЯП вручную требует очень большого объема работы. Для решения этой задачи нужен специализированный инструмент – генератор LR-анализаторов. К счастью, имеется множество таких генераторов, например *yacc* или *bison*. Такой генератор получает КСГ и автоматически строит по ней синтаксический анализатор. Если грамматика содержит неоднозначности или другие конструкции, трудные для синтаксического анализа чтением входного потока слева направо, генератор локализует их и предоставляет детальную диагностическую информацию.

Ситуации и LR(0)-автомат

Каким образом ПС-анализатор выясняет, когда следует выполнять перенос, а когда – свертку? Например, когда стек на СЛАЙДЕ 5 содержит $\$T$, а очередной входной символ – *, каким образом синтаксический анализатор узнает, что *T* на вершине стека – не основа, так что корректное действие – перенос, а не свертка *T* в *E*?

LR-анализатор принимает решение о выборе перенос или свертки, поддерживая

состояния, которые отслеживают, где именно в процессе синтаксического анализа он находится. Состояния представляют собой множества «ситуаций». **LR(0)-ситуация** (далее – просто **ситуация**) грамматики *G* – это продукция *G* с точкой в некоторой позиции RHS. Следовательно, продукция *A -> XYZ* дает четыре ситуации (СЛАЙД 11):

$$A \rightarrow \cdot XYZ$$
$$A \rightarrow X \cdot YZ$$
$$A \rightarrow XY \cdot Z$$
$$A \rightarrow XYZ \cdot$$

Продукция *A -> ε* генерирует единственную ситуацию (*A -> ·*). Как и в случае универсальных методов СА, ситуация указывает, какая часть продукции уже просмотрена в данной точке СА. Первая из показанных ситуаций указывает, что во входном потоке мы ожидаем встретить строку, порождаемую *XYZ*. Вторая ситуация указывает, что нами уже просмотрена строка, порожденная *X*, и мы ожидаем получить из входного потока строку, порождаемую *YZ*. Четвертая ситуация говорит о том, что уже обнаружено *XYZ*, и что, возможно, пришло время свернуть *XYZ* в *A*.

Один набор множеств LR(0)-ситуаций, именуемый **каноническим набором LR(0)**, обеспечивает основу для построения ДКА, который используется для принятия решений в процессе СА (СЛАЙД 12). Такой автомат называется **LR(0)-автоматом**. В частности, каждое состояние LR(0)-автомата представляет множество ситуаций в каноническом наборе LR(0). Автомат для использовавшейся ранее грамматики выражений, показанный на СЛАЙДЕ 13, будет служить в качестве примера при рассмотрении канонического LR(0)-набора грамматики.

Для построения канонического LR(0)-набора мы определяем расширенную (пополненную) грамматику и две функции, *CLOSURE* и *GOTO*. Если *G* – грамматика с аксиомой *S*, то расширенная грамматика *G'* представляет собой *G* с новой аксиомой *S'* и продукцией *S' -> S*. Назначение этой новой аксиомы – указать синтаксическому анализатору, когда следует прекратить анализ и сообщить о принятии входной строки; т.е. принятие осуществляется тогда и только тогда, когда синтаксический анализатор выполняет свертку с использованием продукции *S' -> S*. СЛАЙД 14.

Если *I* – множество ситуаций грамматики *G*, то *CLOSURE*(*I*) представляет собой множество ситуаций, построенное из *I* по двум правилам (СЛАЙД 15).

1. Изначально в *CLOSURE*(*I*) добавляются все элементы *I*.

2. Если *A -> α•Bβ* входит в *CLOSURE*(*I*), а *B -> γ* является продукцией, то в *CLOSURE*(*I*) добавляется ситуация *B -> • γ*, если ее там еще нет. Это правило применяется до тех пор, пока не останется ситуаций для добавления в *CLOSURE*(*I*).

Неформально, *A -> α•Bβ* в *CLOSURE*(*I*) указывает, что в некоторой точке процесса синтаксического анализа мы полагаем, что далее во входной строке мы можем встретить подстроку, порождаемую из *Bβ*. Эта подстрока будет иметь префикс, порождаемый из *B* путем применения одной из *B*-продукций. Таким образом, мы добавляем ситуации для всех *B*-продукций; т.е. если *B -> γ* является продукцией, то мы включаем *B -> • γ* в *CLOSURE*(*I*).

Пример 74.

Рассмотрим расширенную грамматику выражений (СЛАЙД 16).

Если *I* – множество из одной ситуации {[*E'* -> • *E*]}, то *CLOSURE*(*I*) содержит множество, показанное на СЛАЙДАХ 10 (продублировано на СЛАЙДЕ 16).

Рассмотрим, как вычисляется замыкание. *E'* -> *E* помещается в *CLOSURE*(*I*) согласно правилу (1). Поскольку непосредственно справа от точки находится *E*, то добавляются *E*-продукции с точками на левом конце: *E -> • E + T* и *E -> • T*. Теперь справа от точки в последней продукции находится *T*, так что следует добавить *T -> • T * F* и *T -> • F*. Далее, *F* справа от точки заставляет добавить *F -> • (E)* и *F -> • id*, и больше никакие другие

ситуации не добавляются.

Замыкание может быть вычислено так, как показано на СЛАЙДЕ 17. Удобный способ реализации функции *closure* состоит в поддержании булева массива *added*, индексированного нетерминалами грамматики *G*, так что *added*[*B*] устанавливается равным *true*, если и когда мы добавляем ситуацию *B* -> • *γ* для каждой *B*-продукции *B* -> *γ*.

Заметим, что если одна *B*-продукция добавляется в замыкание *I* с точкой на левом конце, то в замыкание будут аналогичным образом добавлены все *B*-продукции. Следовательно, при некоторых условиях нет необходимости в перечислении ситуаций *B* -> • *γ*, добавленных в *I* функцией *CLOSURE*. Достаточно списка нетерминалов *B*, продукции которых были добавлены таким образом. Разделим множество интересующих нас ситуаций на два класса (СЛАЙД 18).

1. **Базисные ситуации**, или **ситуации ядра** (*kernel items*): начальная ситуация *S*' -> • *S* и все ситуации, у которых точки расположены не у левого края.

2. **Небазисные** (*nonkernel items*) **ситуации**, у которых точки расположены слева, за исключением *S*' -> • *S*.

Кроме того, каждое множество интересующих нас ситуаций формируется как замыкание множества базисных ситуаций. Добавляемые в замыкание ситуации не могут быть базисными. Таким образом, мы можем представить множества интересующих нас ситуаций с использованием очень небольшого объема памяти, если отбросим все небазисные ситуации, зная, что (и как) они могут быть восстановлены процессом замыкания. На СЛАЙДЕ 10 были показаны небазисные ситуации, которые размещались в заштрихованных частях прямоугольников состояний.

Второй используемой функцией является *GOTO*(*I*, *X*), где *I* – множество ситуаций, а *X* – символ грамматики. *GOTO*(*I*,*X*) определяется как замыкание множества всех ситуаций [*A* -> *αX* • *β*], таких, что [*A* -> *α* • *Xβ*] находится в *I*.

Неформально, функция *GOTO* используется для определения переходов в LR(0)-автомате грамматики. Состояния автомата соответствуют множествам ситуаций, и *GOTO*(*I*, *X*) указывает переход из состояния *I* при входном символе *X*. СЛАЙД 19.

Пример 75.

Если *I* – множества из двух ситуаций {[*E*' -> *E* •], [*E* -> *E* • + *T*]}, то *GOTO*(*I*, +) содержит ситуации, показанные на СЛАЙДЕ 20.

GOTO(*I*, +) вычисляется путем рассмотрения ситуаций *I*, в которых + следует непосредственно за точкой. *E*' -> *E* • такой ситуацией не является, но ею является ситуация *E* -> *E* • + *T*. Поэтому мы переносим точку за +, получая ситуацию *E* -> *E* + • *T*, а затем находим замыкание этого множества из одного элемента.

Теперь мы готовы к построению канонического набора *C* множеств LR(0)-ситуаций для расширенной грамматики *G*'. Соответствующий алгоритм показан на СЛАЙДЕ 21.

Построив канонический набор, мы можем перейти к созданию LR(0)-автомата, поскольку основная идея, лежащая в основе SLR-анализа, заключается в построении LR(0)-автомата для заданной грамматики. Состояниями этого автомата являются множества ситуаций из канонического набора LR(0), а переходы определяются функцией *GOTO*. LR(0)-автомат для грамматики выражений мы уже приводили ранее.

Стартовое состояние LR(0)-автомата – *CLOSURE*({[*S*' -> • *S*]}), где *S*' – новая аксиома. Все состояния являются принимающими. Под состоянием *j* далее мы понимаем состояние, соответствующее множеству ситуаций I_j.

Каким образом LR(0)-автомат помогает в принятии решения при переносе или свертке? Это решение может быть принято следующим образом. Предположим, что строка *γ* из символов грамматики переводит LR(0)-автомат из состояния 0 в некоторое

состояние *j*. Тогда выполним перенос очередного входного символа *a*, если состояние *j* имеет переход для данного символа *a*. В противном случае выбирается свертка; ситуация в состоянии *j* говорит нам, какую продукцию следует для этого использовать. СЛАЙД 22.

Алгоритм LR-анализа, который приводится далее, использует стек для отслеживания и состояний, и символов грамматики. Фактически же символы грамматики могут быть восстановлены из состояний, так что стек хранит только состояния. Приведенный далее пример показывает, каким образом LR(0)-автомат и стек могут использоваться для принятия решения о действиях в процессе СА.

Пример 76.

На СЛАЙДЕ 23 показаны действия ПС-анализатора для строки *id*id* с использованием нашего LR(0)-автомата. Стек используется для хранения состояний; для ясности в колонке «Символы» приведены символы грамматики, соответствующие состояниям. В строке (1) в стеке находится стартовое состояние 0 автомата; соответствующий ему символ – маркер дна стека $.

Очередной входной символ – *id*, а состояние 0 имеет переход по *id* в состояние 5. Таким образом, выбирается перенос. Во второй строке состояние 5 (символ *id*) вносится в стек. Переходов из состояния 5 для входного символа * нет, так что выбирается свертка. Ситуация [*F -> id* •] в состоянии 5 указывает, что свертка выполняется с использованием продукции *F -> id*.

Что касается символов, то свертка выполняется путем снятия RHS продукции со стека (во второй строке – *id*) и размещения в нем LHS продукции (у нас – *F*). В стеке мы снимаем состояние 5 для символа *id*, что приводит к поднятию состояния 0 на вершину стека, и ищем переходы для *F*, которая является LHS использованной в свертке продукции. У нас состояние 0 имеет переход по *F* в состояние 3, так что в стек помещается состояние 3 с соответствующим символом *F*.

В качестве еще одного примера рассмотрим пятую строку с состоянием 7 (символ *) на вершине стека. Это состояние имеет переход в состояние 5 для входного символа *id*, так что мы помещаем в стек состояние 5 (символ *id*). У состояния 5 нет переходов, поэтому выполняется свертка в соответствии с продукцией *F -> id*. Когда со стека снимается состояние 5, соответствующее телу продукции *id*, на вершине стека оказывается состояние 7. Поскольку состояние 7 имеет переход по символу *F* в состояние 10, мы вносим в стек состояние 10 (символ *F*).

11.3 Алгоритм LR-анализа

Схематически LR-анализатор показан на СЛАЙДЕ 24. Он состоит из входного буфера, выхода, стека, программы-драйвера и таблицы синтаксического анализа, состоящей из двух частей (*ACTION* и *GOTO*). Программа-драйвер одинакова для всех LR-анализаторов; от одного анализатора к другому меняются ТСА. Синтаксический анализатор по одному считывает символы из входного буфера. Там, где ПС-анализатор должен перенести символ, LR-анализатор переносит состояние. Каждое состояние подытоживает информацию, содержащуюся в стеке ниже него.

Стек хранит последовательность состояний $s_0 s_1 \ldots s_m$, где s_m находится на вершине стека. В случае метода SLR стек хранит состояния LR(0)-автомата. Рассматриваемые в следующем разделе лекционного курса канонический метод LR и LALR-метод аналогичны.

В соответствии с построением у каждого состояния есть соответствующий грамматический символ. Ранее уже указывалось, что состояния соответствуют множествам ситуаций, и что 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*. Так, каждое состояние, за исключением стартового состояния 0, имеет единственный грамматический символ, связанный с ним.

Версия 0.9pre-release от 12.05.2014. Возможны незначительные изменения.

ТСА состоит из двух частей: функции действий *ACTION* и функции переходов *GOTO* (СЛАЙД 25).

1. Функция *ACTION* принимает в качестве аргумента состояние *i* и терминал *a* (или маркер конца входной строки). Значение *ACTION*[*i*, *a*] может быть одного из следующих видов.

1а) **Перенос** в состояние *j*. Действие, предпринимаемое синтаксическим анализатором, эффективно переносит входной символ *a* в стек, но для представления *a* использует состояние *j*.

1б) **Свертка** согласно продукции $A \to \beta$. Символы основы заменяются на вершине стека на *A*.

1в) **Принятие**. Синтаксический анализатор принимает входную строку и завершает анализ.

1г) **Ошибка**. Синтаксический анализатор обнаруживает ошибку во входной строке и предпринимает некоторое корректирующее действие.

2) Функция *GOTO*, определенная на множествах ситуаций, распространяется на состояния: если $GOTO[I_i, A] = I_j$, то *GOTO* отображает также состояние *i* и нетерминал *A* на состояние *j*.

Описать поведение LR-анализатора можно с помощью обозначений, представляющих полное состояние синтаксического анализатора: его стек и оставшуюся непроанализированной часть входной строки. Конфигурация LR-анализатора представляет собой пару ($s_0 s_1 \ldots s_m$, $a_i a_{i+1} \ldots a_n$\$).

Здесь первый компонент – содержимое стека, а второй компонент – оставшаяся непроанализированной часть входной строки. Эта конфигурация представляет собой правую СФ $X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n$ по сути, тем же способом, что и ПС-анализатор. Отличие лишь в том, что вместо символов грамматики в стеке хранятся состояния, из которых могут быть восстановлены грамматические символы. Иными словами, X_i является грамматическим символом, представленным состоянием s_i. Стартовое состояние синтаксического анализатора s_0 не представляет символ грамматики, а служит маркером дна стека и играет важную роль в процессе анализа.

Очередной шаг синтаксического анализатора из приведенной выше конфигурации определяется считанным текущим входным символом a_i и состоянием на вершине стека s_m путем обращения к записи *ACTION*[s_m,a_i] в таблице действий синтаксического анализа. СЛАЙД 26.

В результате выполнения указанного действия получаются следующие конфигурации (СЛАЙДЫ 27-29).

1. Если *ACTION*[s_m, a_i] = перенос *s*, то синтаксический анализатор выполняет перенос в стек очередного состояния *s* и его очередной конфигурацией становится ($s_0 s_1 \ldots s_m s$, $a_{i+1} \ldots a_n$\$).

Символ a_i хранить в стеке не нужно, поскольку при необходимости он может быть восстановлен из *s*. Текущим входным символом становится a_{i+1}.

2. Если *ACTION*[s_m, a_i] = свертка $A \to \beta$, то синтаксический анализатор выполняет свертку и его конфигурацией становится ($s_0 s_1 \ldots s_{m-r} s$, $a_i a_{i+1} \ldots a_n$\$).

Здесь *r* – это длина β, а $s = GOTO[s_{m-r}, A]$. Синтаксический анализатор вначале снимает *r* символов состояний с вершины стека, что переносит на вершину стека состояние s_{m-r}, после чего на вершину стека помещается *s*, запись из $GOTO[s_{m-r}, A]$. При свертке текущий входной символ не изменяется. В SLR-анализаторах (впрочем и в других тоже), которые мы будем строить, последовательность символов $X_{m-r+1} \ldots X_m$ всегда соответствует β, правой части продукции свертки.

3. Если *ACTION*[s_m, a_i] = принятие, то синтаксический анализ завершается.

4. Если *ACTION*[s_m, a_i] = ошибка, то синтаксический анализатор обнаруживает ошибку и вызывает подпрограмму восстановления после ошибки.

Версия 0.9pre-release от 12.05.2014. Возможны незначительные изменения.

Далее приводится алгоритм LR-анализа. Все LR-анализаторы ведут себя похожим способом, отличия – в информации в записях полей *ACTION* и *GOTO* ТСА.

Итак, *входом* алгоритма являются входная строка *w* и таблица LR-анализа с функциями *ACTION* и *GOTO* для грамматики *G*.

Выход: если *w* принадлежит *L*(*G*), то шаги сверток восходящего синтаксического анализа *w*; в противном случае – указание об ошибке.

Методика: изначально в стеке синтаксического анализатора находится начальное состояние s_0, а во входном буфере – *w*$. СЛАЙД 30.

Затем синтаксический анализатор выполняет программу, псевдокод которой приведен на СЛАЙДЕ 31.

Пример 77.

На СЛАЙДЕ 32 показаны канонический набор LR(0)-ситуаций и функции ACTION и GOTO для заданной грамматики. Коды действий таковы:

1. s_i означает перенос и размещение в стеке состояния *i*.
2. r_j означает свертку в соответствии с продукцией с номером *j*.
3. *acc* означает принятие.
4. Пустое поле означает ошибку.

Значение *GOTO*[*s*, *a*] для терминала *a* находится в поле *ACTION*, связанном с переносом для входного символа *a* и состояния *s*. Поле *GOTO* дает значения *GOTO*[*s*, *a*] для нетерминалов *A*. Далее мы выясним, каким образом выбираются записи ТСА.

Для входной строки *i* * *i* + *i* последовательность содержимого стека и входной строки показана на СЛАЙДЕ 33. Для ясности показана также последовательность грамматических символов, соответствующая хранящимся в стеке состояниям.

Например, согласно первой строке ТСА LR-анализатор находится в состоянии 0, начальном состоянии без грамматических символов, и с *i* в качестве первого входного символа.

Действие в строке 0 и столбце *i* поля ACTION – *s*5. Оно означает перенос и внесение в стек состояния 5. Во второй строке выполняется внесение в стек символа состояния 5 и удаление *i* из входного потока.

После этого текущим входным символом становится *; действие для состояния 5 и входного символа * – свертка согласно продукции *F* -> *i*. Со стека при этом снимается один символ состояния, и на вершине стека появляется состояние 0.

Поскольку *GOTO*[0, *F*] равно *s*3, в стек вносится состояние 3. При этом получается конфигурация, показанная в третьей строке. Остальные строки получены аналогично.

11.4 Построение таблиц SLR-анализа

SLR-метод построения ТСА хорошо подходит для изучения LR-анализа. Далее таблицы синтаксического анализа, построенные этим методом, будут называться **SLR-таблицами**, а LR-анализатор, использующий SLR-таблицы, – **SLR-анализатором**. Другие методы расширяют SLR-метод путем информации, получаемой предпросмотром входной строки. СЛАЙД 34.

SLR-метод начинается с LR(0)-ситуаций и LR(0)-автомата, т.е. для данной грамматики *G* мы строим ее расширение *G′* с новой аксиомой *S*'. Для *G*' строится канонический набор *C* множеств ситуаций *G*' вместе с функцией *GOTO*.

Затем строятся записи *ACTION* и *GOTO* в ТСА с использованием следующего алгоритма, который требует от нас знания *FOLLOW*(*A*) для каждого нетерминала *A* грамматики (СЛАЙДЫ 35-37).

Итак, *входом* алгоритма построения таблицы SLR-анализа является расширенная грамматика *G*'.

Выход: функции таблицы SLR-анализа *ACTION* и *GOTO* для грамматики *G*'.

Методика: выполняются следующие действия.

1. Строится $C = \{I_0, I_1, ..., I_n\}$ – набор множеств LR(0)-ситуаций для *G*'.

2. Из I_i строится состояние *i*. Действие СА для состояния *i* определяем следующим образом.

а) Если [*A* -> *α* • *aβ*] принадлежит I_i и $GOTO(I_i, a) = I_j$, то устанавливаем *ACTION*[*i*, *a*] = «перенос *j*». Здесь *a* должно быть терминалом.

б) Если [*A* -> *α* •] принадлежит I_i, то устанавливаем *ACTION*[*i*, *a*] = «свертка *A* -> *a*» для всех *a* из *FOLLOW*(*A*). здесь *A* не должно быть равно *S*'.

в) Если [*S*' -> *S* •] принадлежит I_i, то устанавливаем *ACTION*[*i*, $] = «принятие».

При наличии любого конфликта между действиями, возникшего в результате применения указанных правил, делается вывод о том, что грамматика не принадлежит классу SLR (1). В таком случае алгоритм не может построить синтаксический анализатор для данной грамматики.

3. Переходы для состояния *i* строим для всех нетерминалов *A* с использованием следующего правила: если $GOTO(I_i, A) = I_j$, то $GOTO[i, A] = j$.

4. Все записи, не определенные правилами 2 и 3, получают значение «ошибка».

5. Начальное состояние синтаксического анализатора строится из множества ситуаций, содержащего [*S*' -> • *S*].

ТСА из функций *ACTION* и *GOTO*, определенных при помощи данного алгоритма, называется **SLR(1)-таблицей** грамматики *G*. LR-анализатор с использованием SLR(1)-таблицы для *G* называется **SLR(1)-анализатором** *G*, а грамматика, имеющая SLR(1)-таблицу, называется **SLR(1)-грамматикой**. Обычно единица в SLR(1) опускается, поскольку мы не работаем с синтаксическими анализаторами, предпросматривающими более одного символа.

Пример 78.

Попробуем построить ТСА для пополненной грамматики выражений. Канонический набор множеств ситуаций мы уже строили, он показан на СЛАЙДЕ 32. Сначала рассмотрим q_0.

Ситуация *F* -> • (*E*) приводит к записи *ACTION*[0, (] = «перенос 4», а ситуация *F* -> *id* – к записи *ACTION*[0, *id*] = «перенос 5». Прочие ситуации в q_0 действий не дают. Теперь рассмотрим пункты q_1.

Здесь две ситуации. Первая из них (*E*' -> *E* •) приводит к *ACTION*[1, $] = «принятие», а вторая (*E*' -> *E* • + *T*) – к *ACTION*[1, +] = «перенос 6».

Переходим к q_2. Поскольку *FOLLOW*(*E*) = {$, +,)}, то первая ситуация (*E* -> *T* •) приводит к *ACTION*[2, $] = *ACTION*[2, +] = *ACTION*[2,)] = «свертка *E* -> *T*».

Вторая ситуация (*E* -> *T* • * *F*) приводит к *ACTION*[2, *] = «перенос 7».

Дальнейшие рассуждения приведут к ТСА, похожей на ту, что приводилась на СЛАЙДЕ 33. Номера продукций те же, что и порядок, в котором они перечислены в грамматике. СЛАЙД 38.

Пример 79.

Наша последняя грамматика являлась SLR(1). Каждая такая грамматика однозначна, однако существуют однозначные грамматики, не являющиеся SLR(1). Такова, например, грамматика на СЛАЙДЕ 39.

Нетерминалы *L* и *R* можно рассматривать как *l*- и *r*-значение соответственно, а * – как оператор «содержимое». Канонический набор множеств LR(0)-ситуаций для грамматики показан там же.

Рассмотрим множество ситуаций q_2. Вторая ситуация в этом множестве приводит к тому, что *ACTION*[2, =] становится равным «перенос 8». Поскольку *FOLLOW*(*R*) содержит символ ‘=’ (чтобы убедиться в том, что это так, нужно рассмотреть порождение *S* => *L* =>

R =>* R = R), то вторая ситуация устанавливает запись *ACTION*[2, =] равной «свертка R -> L». Поскольку в записи ACTION[2, =] одновременно оказываются и перенос, и свертка, то при входном символе '=' в состоянии 2 наблюдается конфликт «перенос/свертка». На СЛАЙДЕ 39 это выделено цветом.

Наша грамматика не является неоднозначной. Этот конфликт «перенос/свертка» возникает из того факта, что метод построения SLR-анализатора не достаточно мощен, чтобы запоминать достаточный левый контекст для принятия решения о том, какое действие должно быть предпринято синтаксическим анализатором для входного символа '=' при наличии строки, свертываемой в L. Канонический метод и LALR-метод, например, успешно работают с большим набором грамматик, включая нашу грамматику. Заметим, однако, что существуют такие однозначные грамматики, для которых любой метод построения LR-анализатора приводит к таблице действий с наличием конфликтов. К счастью, при разработке реальных языков программирования таких грамматик можно избежать.

11.5 Активные префиксы

Почему LR(0)-автоматы могут использоваться при принятии решений «перенос/свертка»? LR(0)-автомат для грамматики характеризует строки грамматических символов, которые могут находиться в стеке ПС-анализатора грамматики.

Содержимое стека должно быть префиксом правосентенциальной формы. Если в стеке хранится α, а оставшаяся часть входной строки – x, то последовательность сверток должна привести αx в S. В терминах порождений $S \Rightarrow^* \alpha x$.

Однако в стеке могут находиться не все префиксы правосентенциальных форм, поскольку синтаксический анализатор не должен выполнять перенос после основы. Предположим, например,

$E \Rightarrow^* F * i \Rightarrow (E) * i$

Тогда в разные моменты времени в процессе синтаксического анализа в стеке хранятся {(, (E и (E)}, но в нем не должно находиться (E)*, поскольку (E) является основой, которую синтаксический анализатор должен свернуть в F до того, как выполнит перенос *. СЛАЙД 40.

Префиксы правосентенциальных форм, которые могут находиться в стеке ПС-анализатора, называются **активными префиксами** (*viable prefixes*). Они определяются следующим образом: активный префикс является префиксом правосентенциальной формы, не выходящим за пределы правого конца крайней справа основы СФ. В соответствии с этим определением к концу активного префикса всегда можно добавить терминальные символы для получения правосентенциальной формы.

SLR-анализ основан на том факте, что LR(0)-автомат распознает активные префиксы. Мы говорим, что ситуация $A \rightarrow \beta_1 \bullet \beta_2$ **допустима** (*valid*) для активного префикса $\alpha\beta_1$, если существует правое порождение $S' \Rightarrow^* \alpha A w \Rightarrow \alpha\beta_1\beta_2 w$. Вообще говоря, ситуация может быть допустимой для многих активных префиксов. СЛАЙД 41.

Тот факт, что $A \rightarrow \beta_1 \bullet \beta_2$ допустима для $\alpha\beta_1$, многое говорит нам о том, что именно следует выбрать – перенос или свертку – при обнаружении $\alpha\beta_1$ на вершине стека (СЛАЙД 42). В частности, если $\beta_2 \neq \varepsilon$, то это предполагает, что основа еще не полностью перенесена в стек и очередное действие анализатора – перенос. Если $\beta_2 = \varepsilon$, то $A \rightarrow \beta_1$ – основа, и анализатор должен выполнить свертку в соответствии с этой продукцией. Разумеется, две допустимых ситуации могут указать на разные действия для одного и того же активного префикса. Одни из этих конфликтов могут быть разрешены путем просмотра очередного входного символа, а другие придется разрешать специальными методами. Однако не следует считать, что при применении LR-метода для построения ТСА произвольной грамматики могут быть разрешены все конфликты.

Можно легко вычислить множество допустимых ситуаций для каждого активного префикса, который может появиться в стеке LR-анализатора. Основная теорема LR-

анализа гласит, что множество допустимых ситуаций для активного префикса γ в точности равно множеству ситуаций, достижимых в LR(0)-автомате для данной грамматики из начального состояния по пути, помеченному γ. По сути, множество допустимых ситуаций содержит в себе всю полезную информацию, которая может быть собрана из стека. Мы не будем доказывать данную теорему, а приведем соответствующий пример.

Пример 80.

Рассмотрим пополненную грамматику выражений, множества ситуаций и функцию *GOTO* которой мы уже не раз приводили. Очевидно, что строка $E + T^*$ является активным префиксом этой грамматики. Автомат после чтения этой строки будет находиться в состоянии 7. В нем содержатся следующие ситуации: $T \to T * \bullet F$, $F \to \bullet (E)$, $F \to \bullet id$. Все они являются допустимыми для $E + T^*$. Для того чтобы убедиться, что это так, можно рассмотреть следующие три порождения (СЛАЙД 43).

Показанное слева порождение доказывает допустимость первой ситуации, среднее – второй, а та, что справа – третьей. Можно было бы показать, что других допустимых ситуаций здесь нет, но мы оставим это в качестве самостоятельного упражнения.

НКА N для распознавания активных префиксов может быть построен путем рассмотрения ситуаций в качестве состояний. Существует переход из $A \to \alpha \bullet X\beta$ в $A \to \alpha X \bullet \beta$, помеченный X, и переход из $A \to \alpha \bullet B\beta$ в $B \to \bullet \gamma$, помеченный ε. В таком случае $CLOSURE(I)$ для множества ситуаций (или состояний N) I в точности представляет собой ε-замыкание множества состояний НКА. Таким образом, $GOTO(I, X)$ дает переход из I для символа X в ДКА, построенном из N при помощи метода конструирования подмножеств. При таком подходе процедура $items(G')$ представляет собой процедуру построения подмножества, примененную к НКА N, состояниями которого являются ситуации. СЛАЙД 44.

Литература к лекции 11

1. LR(0) - http://ru.wikipedia.org/wiki/LR(0)
2. SLR(1) - http://ru.wikipedia.org/wiki/SLR(1)
3. Ахо, А. Компиляторы: принципы, технологии и инструментарий, 2 издание / А. Ахо, М.Лам, Р. Сети, Дж. Ульман. – М.: Издательский дом «Вильямс», 2008. – 1184 с.
4. LR-анализатор - http://ru.wikipedia.org/wiki/LR
5. The Lex & Yacc Page - http://dinosaur.compilertools.net/