Лекция 12. Синтаксический анализ. Часть 4

Главные вопросы, которые мы обсуждаем, представлены на СЛАЙДЕ 1.

12.1 Канонический LR(1)-анализ

Рассмотрим наиболее общий метод построения ТСА для грамматики. Выше мы показали, что в SLR-методе состояние *i* вызывает свертку в соответствии с продукцией *A -> α*, если множество ситуаций I_i содержит [*A -> α* •], а текущий входной символ *a* входит в *FOLLOW*(*A*). В некоторых ситуациях, однако, когда состояние *i* находится на вершине стека, допустимый префикс *βα* в стеке таков, что за *βα* ни в какой правосентенциальной форме не может следовать *α*. Таким образом, свертка в соответствии с продукцией *A -> α* оказывается некорректной при входном символе *a*.

Можно сохранять в состоянии больший объем информации, который позволит отбрасывать такие некорректные свертки. Разделяя при необходимости состояния, можно добиться того, что каждое состояние LR-анализатора будет точно указывать, какие входные символы могут следовать за основой *α*, для которой возможна свертка в *A*.

Дополнительная информация вносится в состояние путем такого переопределения ситуаций, чтобы они включали в качестве второго компонента терминальный символ. Общим видом ситуации становится [*A -> α • β*, *a*], где *A -> αβ* – продукция, а *a* – терминал или маркер конца строки $. Такой объект называется LR(1)-ситуацией. Здесь 1 означает длину второго компонента, именуемого **предпросмотром** (*lookahead*) ситуации. Предпросмотр не влияет на ситуацию вида [*A -> α • β*, *a*], где *β* не равно *ε*, но ситуация [*A -> α •*, *a*] приводит к свертке в соответствии с продукцией *A -> α*, если очередной входной символ равен *a*. СЛАЙД 2.

Таким образом, свертка в соответствии с продукцией *A -> α* применяется только при входном символе *a*, для которого [*A -> α •*, *a*] является LR(1)-ситуацией из состояния на вершине стека. Множество таких *a* всегда является подмножеством *FOLLOW*(*A*), но может быть истинным подмножеством, как в предыдущем примере.

Считаем, что LR(1)-ситуация [*A -> α • β*, *a*] **допустима** (*valid*) для активного префикса *γ*, если существует порождение $S \underset{rm}{\overset{*}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta\alpha\beta w$, где

1) $\gamma = \delta\alpha$;

2) либо *a* является первым символом *w*, либо *w* = *ε*, а *a* = $.

В грамматике

S -> BB

B -> aB | b

Существует правое порождение $S \underset{rm}{\overset{*}{\Rightarrow}} aaBab \underset{rm}{\Rightarrow} aaaBab$. Мы видим, что ситуация [*B -> a • b*, *a*] *допустима* для активного префикса *γ* = *aaa*, если в приведенном выше определении δ = *aa*, *A* = *B*, *w* = *ab*, *α* = *a* и *β* = *B*.

Существует также правое порождение $S \underset{rm}{\overset{*}{\Rightarrow}} BaB \underset{rm}{\Rightarrow} BaaB$. Из него видно, что ситуация [*B -> a • B*, $] является допустимой для активного префикса *Baa*. СЛАЙД 3.

Метод построения набора множеств допустимых LR (1)-ситуаций, тот же, что и для

построения канонического набора множеств LR(0)-ситуаций. Мы должны модифицировать две процедуры – *CLOSURE* и *GOTO*.

Чтобы разобраться в новом определении операции *CLOSURE* (в частности, почему *b* должно быть в *FIRST* (*β a*)), рассмотрим ситуацию вида [*A* -> *α* • *Bβ*, *a*] в множестве ситуаций, допустимых для некоторого активного префикса *γ*. Тогда существует правое порождение $S \underset{rm}{\overset{*}{\Rightarrow}} \delta Aax \underset{rm}{\Rightarrow} \delta\alpha B\beta ax$, где $\gamma = \delta\ \alpha$. Предположим, что *βax* порождает строку терминалов *βy*. Тогда для каждой продукции вида *B* -> *η* для некоторого *η* мы имеем порождение $S \underset{rm}{\overset{*}{\Rightarrow}} \gamma Bby \underset{rm}{\Rightarrow} \gamma\eta by$. Таким образом, [*B* -> • *Bη*, *b*] является допустимой для *γ*. При этом, что *b* может быть первым терминалом, порожденным из *β*, либо, когда *β* порождает *ε* в порождении $\beta ax \underset{rm}{\overset{*}{\Rightarrow}} by$, то *b* может представлять собой *a*. Подытоживая обе возможности, мы говорим, что *b* может быть любым терминалом в *FIRST*(*βax*), где FIRST – процедура вычисления множества первых порождаемых символов. Заметим, что *x* не может содержать первый терминал из *by*, так что *FIRST*(*βax*) = FIRST(*βa*). СЛАЙД 4.

Приведем теперь алгоритм построения множеств LR(1)-ситуаций (СЛАЙД 5).

ВХОД: расширенная грамматика *G*'.

ВЫХОД: множества LR(1)-ситуаций, которые представляют собой множество ситуаций, допустимых для одного или нескольких активных префиксов *G*'.

МЕТОД: процедуры *CLOSURE* и *GOTO* и основная функция *items* для построения множеств ситуаций приведены на СЛАЙДЕ 5.

Пример 81.

Рассмотрим следующую расширенную грамматику (СЛАЙД 6):

S' -> *S*

S -> *C C*

C -> *cC* | *d*

Начнем с вычисления замыкания {[*S*' -> • *S*, $]}. В процедуре *CLOSURE* мы сопоставляем ситуацию [*S*' -> • *S*, $] с ситуацией [*A* -> *α* • *Bβ*, *a*], т.е. *A* = *S*', *a* = ε, *B* = *S*, *β* = ε и *a* = $. Функция *CLOSURE* говорит нам, что следует добавить ситуацию [*B* -> • *γ*, *b*] для каждой продукции *B* -> *γ* и терминала *b* из FIRST (*βa*). В терминах грамматики B -> *γ* должна быть *S* -> *C C*, а поскольку β = ε, а *a* = $, *b* может быть только $. Таким образом, мы добавляем [*S* -> • *C C*, $].

Продолжим вычисление замыкания путем добавления всех ситуаций [*C* -> • *γ*, *b*] для *b* из *FIRST* (*C*$). Сопоставляем [*S* -> • *C C*, $] с [*A* -> *α* • *Bβ*, *a*] и получаем *A* = *S*, *α* = *ε*, *B* = *C*, *β* = *C* и *a* = $. Поскольку *C* не порождает пустой строки, то *FIRST*(*C*$) = *FIRST*(*C*). Так как *FIRST*(*C*) содержит терминалы *c* и *d*, то мы добавляем ситуации [*C* -> •*cC*, *c*], [*C* -> • *cC*, *d*], [*C* -> • *d*, *c*] и [*C* -> • *d*, *d*]. Для экономии места две первые ситуации запишем более коротко [*C* -> •*cC*, *c* / *d*], а две последние – [*C* -> • *d*, *c* / *d*].

Ни одна из новых ситуаций не имеет нетерминала непосредственно справа от точки, так что первое множество LR(1)-ситуаций I_0 завершено.

Теперь можно вычислить *GOTO*(I_0, *X*) для различных значений *X*. Для *X* = *S* мы должны вычислить замыкание ситуаций [*S*' -> *S* •, $]. Никакие дополнительные замыкания невозможны, поскольку точка располагается крайней справа. Таким образом, мы получаем следующее множество ситуаций I_1.

Для *X* = *C* вычисляем замыкание [*S* -> *C* • *C*, $]. Добавляем *C*-продукции со вторым компонентом $, после чего не можем добавить ничего более и, таким образом, получаем множество I_2.

Далее пусть *X* = *c*, и замыкается {[*C* -> *c* • *C*, *c*/*d*]}. Добавляем *C*-продукции со вторым компонентом *c*/*d*, что дает множество I_3.

И, наконец, для *X* = *d* находим множество I_4 = {[*C* -> *d* •, *c*/*d*] }. На этом построение *GOTO* на множестве I_0 заканчивается. Для I_1 новые множества не образуются, но I_2 имеет переходы по *C*, *c* и *d*. Для нетерминала *C* получаем I_5 = {[*S* -> *C C* •, $]}.

Для терминала *c* рассматривается замыкание {[*C* -> • *cC*, $]}, получаем множество I_6. Видно, что I_6 отличается от I_3 только вторыми компонентами. Вообще, некоторые множества LR(1)-ситуаций для грамматики могут иметь одинаковые первые компоненты и отличаются друг от друга своими вторыми компонентами. При построении наборов множеств LR(0)-ситуаций для той же самой грамматики каждое множество LR(0)-ситуаций совпадает с множеством первых компонентов одного или нескольких множеств LR(1)-ситуаций. Мы воспользуемся этим явлением далее при рассмотрении LALR-анализа.

Продолжая вычисление функции *GOTO* для I_2, получаем, что *GOTO*(I_2,*d*) имеет вид I_7 = {[*C* -> *d* •, $]}.

Просматривая I_3, обнаруживаем переходы из I_3 по *c* и *d*, которые представляют собой соответственно I_3 и I_4, а *GOTO*(I_3, *C*) = I_8. Множества I_4 и I_5 переходов не имеют. Переходы из I_6 по *c* и *d* – это I_6 и I_7. Функций *GOTO*(I_6, *C*) дает множество I_9.

Остальные множества ситуаций не дают значений *GOTO*, так что наша работа завершена. На СЛАЙДЕ 6 показаны найденные десять множеств ситуаций и их переходы.

Теперь приведем правила построения LR(1)-функций *ACTION* и *GOTO* из множеств LR(1)-ситуаций. Эти функции, как и ранее, представлены таблицей. Единственное отличие – в значениях записей таблицы (СЛАЙД 7).

ВХОД: расширенная грамматика *G*'.

ВЫХОД: таблица канонического LR-анализа с функциями *ACTION* и *GOTO* для грамматики *G*'.

МЕТОД: выполнить следующие действия.

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

2. Состояние синтаксического анализатора строится из I_i. Действие СА для состояния *i* определяется следующим образом.

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

б) Если [*A* -> *α* •, *a*] входит в I_i и *A* ≠ *S*', то установить ACTION[*i*, *a*] = «свертка A -> *α*».

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

Если при применении указанных правил **обнаруживаются конфликтующие** действия, грамматика **не принадлежит** классу LR(1).

3. Переходы для состояния *i* строятся 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 (2) и (3), считаются записями «ошибка».

5. Начальное состояние анализатора – состояние, построенное из множества ситуаций, содержащего [*S*' -> *S* •, $].

Таблица СА, образованная функциями действий и переходов, полученными при помощи данного алгоритма, называется **канонической таблицей** LR(1)-анализа. LR-анализатор, использующий эту таблицу, называется **каноническим LR(1)-синтаксическим анализатором**. Если функция действий СА не имеет многократно определенных записей, то соответствующая грамматика называется **LR(1)-грамматикой**. Как и ранее, (1) опускается везде, где оно очевидно из контекста.

Пример 82.

Каноническая ТСА для грамматики с продукциями

S -> *C C*

C -> *cC* | *d*

показана на СЛАЙДЕ 8.

Каждая SLR(1)-грамматика является LR(1)-грамматикой, но канонический LR-

анализатор для SLR(1)-грамматики может иметь большее количество состояний, чем SLR-анализатор для той же грамматики. Наша последняя грамматика является SLR-грамматикой и имеет SLR-анализатор с семью .

12.2 LALR(1)-анализ

Перейдем к методу LALR (**LR с предпросмотром** – *lookahead LR*). Этот метод часто используется на практике из-за того, что получаемые при его использовании ТСА получаются сравнительно небольшими по сравнению с каноническими LR-таблицами, а также поскольку большинство распространенных синтаксических конструкций языков программирования могут быть легко выражены LALR-грамматикой. Практически то же самое можно сказать и о SLR-грамматиках, но в этом случае имеется несколько конструкций, которые не могут быть легко обработаны при помощи метода SLR (в этом мы могли убедиться ранее). СЛАЙД 9.

Пример 83.
Рассмотрим еще раз грамматику со СЛАЙДА 6 со следующими продукциями:
S' -> *S*
S -> *C C*
C -> *cC* | *d*

Возьмем пару похожих состояний, таких как I_4 и I_7. Эти состояния содержат только ситуации с первым компонентом *C* -> *d* •. В I_4 предпросматриваемыми символами могут быть *c* и *d*, а в I_7 – только $. СЛАЙД 10.

Чтобы увидеть разницу между I_4 и I_7 в синтаксическом анализаторе, достаточно обратить внимание, что грамматика порождает РЯ, описанный выражением c^*dc^*d. При считывании входного потока *cc...cdcc...cd* анализатор переносит первую группу символов *c* и следующий за ними *d* в стек, попадая после считывания символа *d* в состояние 4. Затем он вызывает свертку по продукции *C* -> *d* обусловленную следующим входным символом *c* или *d*. Требование, чтобы следующим входным символом был *c* или *d*, имеет смысл, поскольку это символы, с которых может начинаться строка c^*d. Если после первого *d* следует $, то получается входной поток наподобие *ccd*, который не принадлежит рассматриваемому языку, и состояние 4 совершенно справедливо обнаруживает ошибку при очередном входном символе $.

В состояние 7 синтаксический анализатор попадает после чтения второго *d*.

Соответственно, синтаксический анализатор должен обнаружить во входном потоке символ $, иначе входная строка не соответствует РВ c^*dc^*d. Таким образом, состояние 7 должно приводить к свертке *C* -> *d* при входном символе $ и к ошибке при входном символе *c* или *d*.

Мы можем рассматривать множества LR(1)-ситуаций, имеющих одно и то же **ядро** (*core*), т.е. множество первых компонентов, и объединить эти множества с общими ядрами в одно множество ситуаций. У нас такую пару с ядром [*C* -> *d* •] образуют состояния I_4 и I_7. Интересно отметить, что ядро является множеством LR(0)-ситуаций рассматриваемой грамматики и что LR(1)-грамматика может давать более двух множеств ситуаций с одним и тем же ядром.

Поскольку ядро множества *GOTO*(*I*, *X*) зависит только от ядра множества *I*, значения функции *GOTO* объединяемых множеств также могут быть объединены. Таким образом, проблем вычисления функции *GOTO* при слиянии множеств не возникает. Функция же *ACTION* должна быть изменена, чтобы отражать не ошибочные действия всех объединяемых множеств ситуаций.

Доказано, что 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).

Рассмотрим алгоритм построения ТСА для LALR-метода. Основная идея состоит в создании множеств LR(1)-ситуаций и, если это не вызывает конфликтов, объединении множеств с одинаковыми ядрами. Затем на базе набора множеств ситуаций строим ТСА.

Описываемый метод служит, в первую очередь, определением LALR(1)-грамматик. Построение полного набора множеств LR(1)-ситуаций требует слишком много памяти и времени, чтобы использоваться на практике (СЛАЙД 11).

ВХОД: расширенная грамматика *G*'.

ВЫХОД: таблица LALR-анализа с функциями *ACTION* и *GOTO* для грамматики *G*'.

МЕТОД: выполнить следующие действия.

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

2. Для каждого ядра, имеющегося среди множества LR(1)-ситуаций, находим все множества, имеющие это ядро, и заменяем эти множества их объединением.

3. Пусть $C' = \{J_0, J_1, ..., J_m\}$ – полученные в результате множества LR (1)-ситуаций. Функцию ACTION для состояния *i* строим из J_i.

а) Если $[A \to \alpha \bullet a\beta, b]$ входит в J_i и $GOTO(J_i, A) = J_j$, то установить ACTION[*i*, *a*] = «перенос *j*». Здесь *a* должно быть терминалом.

б) Если $[A \to \alpha \bullet, a]$ входит в J_i и $A \neq S'$, то установить ACTION[*i*, *a*] = «свертка A -> *α*».

в) Если [*S*' -> *S* •, $] входит в J_i, то установить ACTION[*i*, $] = «принятие».

Если при этом обнаруживается конфликт, алгоритм не в состоянии построить анализатор, а грамматика не является LALR (1)-грамматикой.

4. Таблицу GOTO строим следующим образом. Если *J* – это объединение одного или нескольких множеств LR(1)-ситуаций, т.е. $J = I_1 \cup I_2 \cup ... \cup I_k$, то ядра множеств $GOTO(I_1, X)$, $GOTO(I_2, X)$, ..., $GOTO(I_k, X)$ одни и те же, т.к. $I_1, I_2, ..., I_k$ имеют одно и то же ядро. Обозначим через *K* объединение всех множеств ситуаций, имеющих то же ядро, что и $GOTO(I_1, X)$. Тогда $GOTO(J, X) = K$.

Полученная ТСА называется **таблицей LALR-анализа** для грамматики *G*. Если конфликты действий отсутствуют, то данная грамматика называется **LALR(1)-грамматикой**. Набор множеств ситуаций, построенный на шаге 3, называется **LALR(1)-набором**.

Пример 84.

Опять рассмотрим грамматику со СЛАЙДА 10:

S' -> *S*

S -> *C C*

C -> *cC* | *d*

Функция *GOTO* графически показана на СЛАЙДЕ 12. Имеется три пары множеств ситуаций, которые могут быть объединены. Множества I_3 и I_6 могут быть объединены в одно множество I_{36} с ядром {*C* -> *c* • *C*, *C* -> • *c C*, C -> • *d*}. СЛАЙД 13.

Множества I_4 и I_7 могут быть объединены в одно множество I_{47} с ядром {*C* -> *d* •}. СЛАЙД 14.

Множества I_8 и I_9 могут быть объединены в одно множество I_{89} с ядром {*C* -> *c C* •}. СЛАЙД 15.

Функции действий и переходов LALR для объединенных множеств ситуаций показаны на СЛАЙДЕ 16.

Чтобы увидеть, каким образом вычисляется функция *GOTO*, рассмотрим $GOTO(I_{36},C)$. В исходном множестве LR(1)-ситуаций $GOTO(I_3,C) = I_8$, а I_8 теперь является частью I_{89}, так что мы определяем, что $GOTO(I_{36},C) = I_{89}$. Тот же вывод мы делаем, рассматривая I_6 – вторую часть I_{36}: $GOTO(I_6,C) = I_9$, а I_9 теперь также является частью I_{89}. В качестве другого примера рассмотрим $GOTO(I_2,c)$, переход, выполняемый после переноса

состояния I_2 при входном символе c. В исходных множествах LR(1)-ситуаций $GOTO(I_2, c)$ – I_6. Поскольку теперь I_6 является частью I_{36}, то $GOTO(I_2, c)$ становится равным I_{36}; таким образом, запись в ТСА на СЛАЙДЕ 16 для состояния 2 и входного символа c – s_{36}, что означает перенос и помещение в стек состояния 36.

Работа приведенного алгоритма в сравнении с каноническим LR-анализом обсуждается на СЛАЙДАХ 17 и 18.

Пример 85.

На СЛАЙДЕ 19 продемонстрирована вероятность возникновения конфликта свертка/свертка при объединении ядер множеств состояний.

Эффективное построение ТСА LALR-анализа

Имеется несколько изменений, которые можно внести в обсуждавшийся алгоритм во избежание построения полного набора множеств LR(1)-ситуаций в процессе создания таблицы LALR(1)-анализа.

Во-первых, можно представить любое множество LR(0)- или LR(1)-ситуаций I его ядром, т.е. теми ситуациями, которые либо являются стартовыми ([S' -> • S] или [S' -> • S, \$]), либо содержат точку не в начале тела продукции.

Во-вторых, можно строить ядра LALR(1)-ситуаций на основе ядер LR(0)-ситуаций при помощи процесса распространения («пропагации») и спонтанной генерации символов предпросмотра, которая обсуждается ниже.

В-третьих, если имеются ядра LALR(1), то можно сгенерировать LALR(1)-таблицу путем замыкания каждого ядра с использованием функции *CLOSURE* из знакомого нам алгоритма в левой части СЛАЙДА 20, а затем вычислить записи таблицы при помощи другого знакомого алгоритма в правой части СЛАЙДА 20, как если бы LALR(1)-множества ситуаций были каноническими LR(1)-множествами ситуаций.

Пример 86.

Используем в качестве примера грамматику на СЛАЙДЕ 21. Она не является SLR-грамматикой, и здесь исходная грамматика пополнена одной продукцией. Ядра ситуаций показаны на СЛАЙДЕ 21.

Теперь для создания ядер множеств LALR(1)-ситуаций требуется назначить корректные предпросматриваемые символы ядрам LR(0)-ситуаций. Есть два способа, которыми предпросматриваемый символ b может быть назначен LR(0)-ситуации B -> γ • δ из некоторого множества LALR(1)-ситуаций J. СЛАЙДЫ 22-23.

Нам требуется определить спонтанно генерируемые символы предпросмотра для каждого множества LR(0)-ситуаций, а также определить, для каких ситуаций происходит пропагация предпросмотров и из каких именно ситуаций. Это делается сравнительно легко. Пусть # – символ, отсутствующий в рассматриваемой грамматике, и пусть A -> α • β – ядро LR(0)-ситуации во множестве I. Для каждого грамматического символа X вычислим $J = GOTO(CLOSURE(\{[A \text{ -> } \alpha \bullet \beta, \#]\}), X)$. Для каждой базисной ситуации в J проверим ее множество символов предпросмотра. Если # является символом предпросмотра, то предпросмотры распространяются к этой ситуации от A -> α • β. Все прочие предпросмотры генерируются спонтанно. Эти идеи строго изложены в приведенном далее алгоритме, который, кроме того, использует тот факт, что только в базисных ситуациях J точка непосредственно следует за X. Значит, они должны иметь вид B -> γX • δ. СЛАЙД 24.

Алгоритм определения символов предпросмотра приведен на СЛАЙДЕ 25.

Теперь мы готовы к назначению предпросмотров ядрам множеств LR(0)-ситуаций для формирования множеств LALR(1)-ситуаций. Мы знаем, что \$ является символом предпросмотра для S' —> • S в исходном множестве LR(0)-ситуаций. После того как будут

определены и перечислены все спонтанно генерируемые символы предпросмотра, мы должны позволить им распространяться до тех пор, пока не останется ни одного возможного распространения. Существует много различных подходов, которые в определенном смысле отслеживают «новые» символы предпросмотра, которые распространились на некоторую ситуацию, но пока что не распространились далее. Приведенный на СЛАЙДЕ 26 алгоритм описывает метод распространения символов предпросмотра на все ситуации. Его шаги приведены далее.

1. Строим ядра множеств LR(0)-ситуаций для *G*. Если объем используемой памяти не является главным, то простейший метод состоит в построении LR(0)-множеств ситуаций с последующим удалением небазисных элементов. Можно вместо этого для каждого множества сохранять только базисные ситуации и вычисляются *GOTO* для множества ситуаций *I*, сначала вычисляя замыкание *I*.

2. По ядру каждого множества LR(0)-ситуаций и грамматическому символу *X* определяем, какие символы предпросмотра спонтанно генерируются для базисных ситуаций в *GOTO*(*I*,*X*) и из каких ситуаций *I* символы предпросмотра распространяются на базисные ситуации *GOTO*(*I*,*X*).

3. Инициализируем таблицу, которая для каждой базисной ситуации в каждом множестве дает связанные с ними предпросмотры. Изначально с каждой ситуацией связаны только те предпросмотры, которые в п. 2 определены как сгенерированные спонтанно.

4. Повторим проходы по базисным ситуациям во всех множествах. При посещении ситуации *i* мы с помощью таблицы, построенной в п. 2, ищем базисные ситуации, на которые *i* распространяет свои предпросмотры. Текущее множество предпросмотров для *i* добавляется к множествам, связанным с каждой из ситуаций, на которые *i* распространяет свои предпросмотры. Мы продолжаем выполнять такие проходы по базисным ситуациям до тех пор, пока не останется новых предпросмотров для пропагации.

Пример 87.

Построим ядра LALR(1)-ситуаций для грамматики на СЛАЙДЕ 27. Ядра LR(0)-ситуаций были вычислены ранее. Применяя обсуждавшийся выше алгоритм к ядру множества ситуаций I_0, мы сначала вычисляем замыкание от стартовой ситуации (СЛАЙД 27).

Среди ситуаций в этом замыкании имеются два, у которых символ предпросмотра ‘=’ генерируется спонтанно. Первая из них – *L* -> • * *R*. У нее справа от точки находится ‘*’, и она приводит к [*L* -> * • *R*, =], т.е. ‘=’ – спонтанно сгенерированный символ предпросмотра для *L* -> * • *R*, представляющего собой множество ситуаций I_4. Аналогично [*L* -> • *id*, =] указывает на то, что ‘=’ – спонтанно сгенерированный символ предпросмотра для *L* -> *id* • из I_5.

Поскольку ‘#’ является символом предпросмотра для всех шести ситуаций в замыкании, то мы определяем, что ситуация *S'* -> • *S* в I_0 распространяет символы предпросмотра на шесть ситуаций, показанных на СЛАЙДЕ 27.

На СЛАЙДЕ 28 показаны шаги 3 и 4 алгоритма эффективного вычисления ядер. Столбец «Изначально» указывает спонтанно сгенерированные символы предпросмотра для каждой базисной ситуации. Здесь только два раза встречается рассмотренный ранее символ ‘=’ и спонтанно сгенерированный символ предпросмотра ‘$’ для начальной ситуации *S'* -> • *S*.

При первом проходе предпросмотр $ распространяется от *S'* -> • *S* в I_0 к шести ситуациям, приведенным на СЛАЙДЕ 27. Предпросмотр ‘=’ распространяется от *L* -> * • *R* в I_4 к ситуациям *L* -> * *R* • в I_7 и к *R* -> *L* • в I_8. Кроме того, он также распространяется к самому себе и к *L* -> *id* • из I_5, но эти предпросмотры уже есть. При втором и третьем проходах единственный распространяемый предпросмотр – $ (для I_2 и I_4 обнаруживается при втором проходе и для I_6 – при третьем). При четвертом проходе

пропагации не происходит, и окончательное множество предпросмотров показано в последнем столбце на СЛАЙДЕ 28.

Отметим, что конфликт «перенос/свертка», обнаруженный при использовании SLR-метода, исчезает при использовании LALR-технологии. Причина в том, что с $R \rightarrow L \bullet$ в I_2 связан только предпросмотр \$, так что нет конфликта с переносом '=', генерируемым ситуацией $S \rightarrow L \bullet = R$ из I_2.

12.3 Восстановление после ошибок при LR-анализе

LR-анализатор обнаруживает ошибку при обращении к *ACTION*-части ТСА и нахождении в ней записи об ошибке (при обращении к *GOTO* ошибки не выявляются). LR-анализатор сообщит об ошибке, как только не сможет обнаружить корректного продолжения уже проанализированной части входного потока. Канонический LR-анализ в этом случае даже не выполнит ни одной свертки перед объявлением об ошибке; SLR- и LALR-анализаторы могут произвести ряд сверток перед тем, как сообщить об ошибке, но никогда не будут переносить в стек неверный символ.

При LR-анализе восстановление после ошибок «в режиме паники» обычно реализуется следующим образом. Стек просматривается от вершины до состояния s с записью в *GOTO*-части таблицы для некоторого нетерминала A.

После этого пропускается нуль или несколько символов входного потока, пока не будет найден символ a, который при отсутствии ошибки может следовать за нетерминалом A. Затем анализатор переносит в стек состояние $GOTO(s, A)$ и продолжает выполнение синтаксического анализа. При выборе нетерминала A возможны несколько вариантов. Обычно это нетерминалы, представляющие крупные фрагменты программы, такие как выражение, оператор или блок. Например, если A – нетерминал *stmt*, то a может быть символом ';' или '}', помечающим конец оператора. СЛАЙД 29.

При таком методе восстановления производится попытка выделить фразу, содержащую синтаксическую ошибку. Синтаксический анализатор определяет, что строка, порождаемая из A, содержит ошибку. Часть этой строки уже обработана, и результат этой обработки – последовательность состояний, находящаяся на вершине стека. Остаток строки все еще находится во входном потоке, и синтаксический анализатор пытается пропустить этот остаток, находя символ, который может следовать за A в корректной программе. Удаляя состояния из стека, пропуская часть входного потока и помещая в стек $GOTO(s, A)$, синтаксический анализатор полагает, что им найден экземпляр A, и продолжает обычный процесс анализа.

Восстановление на уровне фразы реализуется путем проверки каждой ошибочной записи в таблице LR-анализа и принятия решения (на основе знания особенностей языка) о том, какая наиболее вероятная ошибка могла привести к данной ситуации. После этого можно построить подходящую функцию восстановления после ошибки; возможно, при этом придется изменить вершину стека и/или первые символы входного потока способом, соответствующим данной записи ошибки. СЛАЙД 30.

При разработке функций обработки ошибок для LR-синтаксического анализатора можно заполнить каждую пустую запись таблицы действия указателем на подпрограмму, которая будет выполнять действия, выбранные для данного случая разработчиком транслятора. Эти действия могут включать вставку символов в стек и/или входной поток и удаление их оттуда или изменение и перестановку входных символов. Выбор должен делаться таким образом, чтобы исключить возможность зацикливания LR-анализатора. Безопасная стратегия должна гарантированно удалять или переносить при каждом цикле по крайней мере один символ из входного потока или при достижении его конца гарантированно уменьшать стек на каждой итерации. Снятия со стека состояния над нетерминалом следует избегать, поскольку такое изменение удаляет из стека успешно разобранную конструкцию.

Пример 88.

В качестве примера еще раз обратимся к грамматике выражений (СЛАЙД 31). Там же показаны две таблицы LR-анализа для этой грамматики: исходную и дополненную функциями обработки ошибок.

Для каждого состояния, вызывающего свертку при том или ином входном символе, все записи ошибок заменены записями некоторых сверток. Такая замена приводит к отложенному обнаружению ошибок после выполнения одной или нескольких лишних сверток; ошибка в любом случае будет найдена до выполнения первого переноса. Оставшиеся пустыми ячейки заполнены указателями на подпрограммы обработки ошибок.

Далее приведены описания подпрограмм обработки ошибок (СЛАЙДЫ 32-35).

Для ошибочной входной строки *id* +) последовательность конфигураций синтаксического анализатора показана на СЛАЙДЕ 36.

12.4 Использование неоднозначных грамматик при LR-анализе

Тот факт, что каждая неоднозначная грамматика не может быть LR-граммтикой и, следовательно, не может относиться ни к одному из рассмотренных ранее классов грамматик, формально доказано, но очевидного доказательства мы приводить не будем.

Ряд типов неоднозначных грамматик, однако, вполне пригоден для определения и реализации языков, как мы увидим далее. Для языковых конструкций наподобие арифметических выражений неоднозначные грамматики обеспечивают более краткую и естественную спецификацию по сравнению с эквивалентными однозначными грамматиками. Другое использование неоднозначных грамматик состоит в выделении распространенных синтаксических конструкций для специализированной оптимизации. Имея неоднозначную грамматику, можно определить специализированные конструкции путем аккуратного добавления в грамматику новых продукций.

Хотя используемые грамматики являются неоднозначными, во всех случаях задаются специальные правила разрешения неоднозначности, обеспечивающие для каждого предложения только одно дерево разбора. В этом смысле полное описание языка остается однозначным, так что иногда оказывается возможной разработка LR-анализатора, который следует тем же правилам разрешения неоднозначности. С другой стороны, неоднозначные конструкции должны использоваться как можно реже и предельно аккуратно; в противном случае нет никакой гарантии, что синтаксический анализатор распознает соответствующий язык.

Пример 89.
На СЛАЙДЕ 37 приведен пример одной такой неоднозначной грамматики:
*E -> E + E | E*E | (E) |* ***id***

Там же показана эквивалентная однозначная грамматика. Она придает оператору + более низкий приоритет по сравнению с оператором * и делает оба оператора левоассоциативными. Имеется две причины, по которым использование неоднозначной грамматики может оказаться предпочтительнее. Во-первых, как мы увидим ниже, можно легко изменить ассоциативность и уровень приоритета операторов + и * без изменения продукций или количества состояний получающегося синтаксического анализатора. Во-вторых, синтаксический анализатор для однозначной грамматики будет тратить значительную часть времени на свертку по продукциям *E -> T* и *T -> F*, единственное назначение которых – определять приоритеты и ассоциативность операторов.

Множества LR(0)-ситуаций для нашей неоднозначной грамматики выражений, расширенной продукцией *E' –> E*, показаны на СЛАЙДЕ 37. Поскольку грамматика неоднозначна, при попытке построить таблицу LR-анализа по этому множеству возникают конфликты действий. В частности, такие конфликты порождают состояния, соответствующие множествам ситуаций I_7 и I_8. Предположим, что мы используем SLR-

метод для построения ТСА. Конфликт порождается множеством I_7 – между сверткой по продукции $E \rightarrow E + E$ и переносом + или *. Он не может быть разрешен, поскольку и +, и * принадлежат множеству *FOLLOW*(*E*). Иначе говоря, для входных символов + и * могут выполняться оба действия. Подобный конфликт порождается множеством I_8 – между сверткой $E \rightarrow E * E$ и переносом при входных символах + и *.

Фактически эти конфликты порождаются при любом способе построения таблиц LR-анализа, а не только при SLR-подходе.

Однако эти проблемы могут быть решены с использованием информации о приоритетах и ассоциативности операторов + и *. Рассмотрим входную строку *id* + *id* * *id*, которая вынуждает синтаксический анализатор попасть в состояние I_7 после обработки *id* + *id*; в частности, синтаксический анализатор достигает конфигурации:

```
Префикс     Стек       Вход
E+E         0 1 4 7    * id $
```

Если приоритет оператора * выше приоритета +, то мы знаем, что анализатор должен выполнить перенос * в стек, подготавливая свертку * и соседних с этим оператором *id* в выражение. Эти же действия будет совершать и SLR-анализатор, основанный на однозначной грамматике для того же языка. С другой стороны, если приоритет + выше приоритета *, то анализатор должен свернуть $E + E$ к E. Таким образом, относительное старшинство +, за которым следует *, однозначно определяет, каким образом должен быть разрешен конфликт между сверткой $E \rightarrow E + E$ и переносом * в состоянии I_7.

Если входная строка имеет вид *id* + *id* + *id*, то синтаксический анализатор после обработки части строки *id* + *id* достигнет конфигурации, при которой содержимое стека – 0 1 4 7. При очередном входном символе + в состоянии I_7 вновь возникает конфликт переноса/свертки. Однако теперь способ разрешения возникшего конфликта определяется ассоциативностью оператора +. Если этот оператор левоассоциативен, то корректным действием анализатора будет свертка. Первыми должны быть сгруппированы *id*, окружающие первый оператор +. И этот выбор совпадает с действиями SLR-анализатора для однозначной грамматики.

В итоге, если оператор + левоассоциативен, то действие в состоянии I_7 для входного символа + должно приводить к свертке по продукции $E \rightarrow E + E$, а предположение о превосходстве приоритета оператора * над + ведет к переносу входного символа *. Аналогично, если оператор * левоассоциативен и старше оператора +, то можно доказать, что состояние I_8, появляющееся на вершине стека только в том случае, когда три верхних символа представляют собой $E * E$, должно приводить к свертке по продукции $E \rightarrow E * E$ как при очередном входном символе +, так и при *. В случае входного символа + причина этого заключается в том, что оператор * старше оператора +, а в случае входного символа * – что оператор * левоассоциативен.

Продолжая рассмотрение, можно получить ТСА, показанную на СЛАЙДЕ 37. Продукции 1-4 представляют собой правила непополненной исходной грамматики. Подобная ТСА может быть получена путем устранения сверток согласно продукциям $E \rightarrow T$ и $T \rightarrow F$ таблицы для однозначной грамматики. В контексте LALR- и канонического LR-анализа с неоднозначными грамматиками такого типа можно поступать аналогично.

Пример 90.

В качестве второго примера обратимся к грамматике для условных инструкций, приведенной на СЛАЙДЕ 38. Эта грамматика неоднозначна, поскольку не разрешает неоднозначности «кочующего *else*». Для упрощения рассмотрим абстрактное представление приведенной грамматики, где символ *i* заменяет *if expr then*, символ *e* – *else*, символ *a* – «все остальные продукции». Наша грамматика, пополненная продукцией

S' –> *S*, может быть записана в том виде, который показан на СЛАЙДЕ 38.

Множества LR(0)-ситуаций показаны на СЛАЙДЕ 38. Неоднозначность грамматики приводит к конфликту переноса/свертки в I_4. Здесь ситуация *S* –> *i S* • *e S* приводит к переносу *e*, а поскольку *FOLLOW*(*S*) = {*e*, $}, ситуация *S* -> *iS* • приводит к свертке по продукции *S* -> *iS* при входном символе *e*.

Если перевести это обратно в термины *if-then-else*, то, при наличии в стеке *if expr then stmt* и *else* в качестве первого входного символа, должен ли анализатор перенести *else* в стек (т.е. выполнить перенос *e*) или свернуть *if expr then stmt* в *stmt* (т.е. выполнить свертку по продукции *S* -> *iS*). Ответ заключается в том, что мы должны перенести *else*, так как оно «связано» с предыдущим *then*. В терминах грамматики входной символ *e* может быть только частью тела продукции, начинающегося с части *iS*, которая находится на вершине стека.

В итоге, мы можем заключить, что конфликт перенос/свертка в I_4 должен быть разрешен в пользу переноса входного символа *e*. Таблица SLR-анализа, построенная по множествам ситуаций с использованием описанного разрешения конфликта, показана на СЛАЙДЕ 38. Здесь продукции 1-3 представляют собой продукции исходной грамматики.

При входной строке *iiaea* анализатор выполняет действия, показанные на СЛАЙДЕ 30, в соответствии с корректным разрешением проблемы «кочующего *else*». В строке (5) в состоянии 4 происходит перенос входного символа *e*, а в строке (9) в состоянии (4) при входном $ выполняется свертка по продукции *S* -> *i S*. В случае, когда мы не можем использовать неоднозначную грамматику для условных инструкций, можно воспользоваться более громоздкой грамматикой на СЛАЙДЕ 39.

12.5 LR(k)-грамматики, GLR-анализ, IELR-анализ

Пример 91.

На СЛАЙДЕ 40 приведен пример LR(2)-грамматики. Данная грамматика не является LR(1), т.к. при считывании фрагмента *ax* и предпросмотре символа “,” неизвестно, что применять: свертку по продукции 4 или перенос, как в продукции 3. Два символа предпросмотра решают конфликт, т.к. из предпросмотра пары “, x” следует перенос, а из предпросмотра “, a” – свертка. Это объясняется тем, что символы “, S” являются последователя *L* согласно продукции 1, а “a” – стартовый символ для *S*, также согласно 1 продукции. Кроме того, “x” – это стартовый символ для нетерминала *L*.

Формально доказано, что для любой LR(*k*)-грамматики с $k > 1$ можно найти эквивалентную LR(1)-грамматику, поэтому в большинстве случаев достаточно одного символа предпросмотра.

Помимо LALR-алгоритма можно воспользоваться методом GLR, а также IELR(1)-алгоритмом. СЛАЙД 41.

Литература и дополнительные источники к Лекции 12

1. Backus, J.W. Revised Report on the Algorithmic Language ALGOL 60. / J.W. Backus et al., in P. Naur editions. // Commun. ACM, vol. 6, no. 1, p. 1-17, January 1960
2. Standard ECMA-262. ECMAScript Language Specification – http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf
3. Standard ECMA-334. C# Language Specification – http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf
4. Grune, D. Parsing Techniques: a practical guide. / D. Grune, C. Jakobs. – Ellis Horwood, Chichester, 1990. – 334 p
5. Knuth, D. E. Semantics of context-free languages/ D.E. Knuth // Mathematical Systems Theory 2:2, 1968. – P. 127-145
6. Wirth, N. The design of a Pascal compiler. / N. Wirth // Software – Practice and Experience 1:4, 1971, p. 309-333
7. Хомский, Н. «Три модели для описания языка». / Н. Хомский // Кибернетический

сборник. – М. ИЛ, 1961. – Вып. 2. – С. 237-266
8. Мозговой, М.В. Классика программирования: алгоритмы, языки, автоматы, компиляторы Практический подход / М.В. Мозговой. – СПб.: Наука и техника, 2006. – 320 с.
9. Гросс, М. Теория формальных грамматик. / М. Гросс, А. Лантен. – М.: Мир, 1971. – 296 с.
10. Waite, W.M. Compiler Construction / W.M. Waite, G. Goos. – Berlin: Springer Verlag, 1996. – 372 p.