Лекция 9. Синтаксический анализ. Часть 1

Главные вопросы, которые мы обсуждаем, представлены на СЛАЙДЕ 1. Сначала мы обсуждаем задачу и известные стратегии синтаксического анализа. Далее мы рассматриваем два универсальных алгоритма синтаксического разбора.

9.1 Общая задача синтаксический анализ

Имея грамматику, мы можем получить порождение какой-либо терминальной СФ. Задача синтаксического анализа – обратная. По заданной терминальной СФ нужно восстановить ее порождение из аксиомы грамматики.

СА можно понимать и как восстановление дерева разбора заданной входной строки. Для дерева известны листья (входная строка) и корень (аксиома грамматики). Рассуждения о восстановлении дерева эквивалентны рассуждениям о восстановлении порождения входной строки из аксиомы, т.е. восстановление каждого шага порождения соответствует добавлению очередного узла в дерево и обратно. СЛАЙД 2.

9.2 Нисходящий и восходящий синтаксический анализ

В терминах восстановления дерева разбора задача СА представлена на СЛАЙДЕ 3. Ее можно решать, используя две различные стратегии. Можно восстанавливать дерево от корня к листьям. Эта стратегия получила название СА сверху вниз, или **нисходящего** СА (*top-down parsing*, далее – НСА). Можно восстанавливать дерево от листьев к корню. Это синтаксический анализ снизу вверх, или **восходящий** СА (*bottom-up parsing*, далее – ВСА). СЛАЙД 4.

НСА работает так, как показано на СЛАЙДЕ 5. Начиная от корня дерева, мы пытаемся найти те продукции грамматики, которые нужно использовать для подстановки RHS продукций вместо каждого нетерминала, стоящего в узлах дерева. Основным повторяющимся вопросом является следующий. Какую из возможных альтернатив для данного нетерминала выбрать, чтобы в результате получить заданную терминальную строку? Критерием выбора является, конечно, совпадение порожденной и заданной строк, Это может стать ясным только после выполнения всех подстановок. Однако на промежуточных шагах можно проверять совпадение терминального префикса получающейся СФ и префикса той же длины заданной терминальной строки. Несовпадение префиксов указывает на ошибку в выборе альтернативы на одном из предыдущих шагов, и требует возврата в алгоритме.

Рассмотрим наиболее простой – **переборный** алгоритм, он же – алгоритм **грубой силы** (*Brute force*). Пусть задана грамматика G с продукциями $S \to abSc \mid bA$, $A \to ab \mid cBA$, $B \to bBc \mid c$.

На СЛАЙДЕ 5 показаны два шага этого алгоритма.

На первом шаге для единственного нетерминала S – корня дерева из двух альтернатив выбираем первую. Это дает промежуточную СФ $abSc$, которая после дальнейших подстановок должна совпасть с заданной строкой $abbccabc$. Максимальный левый терминальный префикс (ab) этой промежуточной СФ совпадает с соответствующим префиксом заданной строки, что указывает на возможно правильное сделанное предположение о том, что сделанный шаг алгоритма не противоречит требованиям алгоритма. Следующий шаг также заключается в выборе одной из альтернатив для S так, чтобы из Sc можно было получить строку $bccabc$ – остаток исходной строки. Выбор

первой альтернативы дает СФ *abScc*, но сравнение терминальных префиксов указывает на неправильный выбор альтернативы либо на данном, либо на каком-то из предыдущих шагов. Результатом является **откат** (или **бэктрекинг**) на один шаг назад для попытки выбора другой альтернативной ветки.

Откат неудобен по разным причинам. В частности, такие алгоритмы требуют большого времени исполнения. Чтобы избежать отката при выполнении НСА, для заданной грамматики необходимо сформулировать однозначный критерий выбора альтернативы для каждого нетерминала A в каждом случае, когда он стоит в начале некоторой СФ $A\gamma$, из которой следует породить некоторую терминальную СФ β – остаток исходной строки. В общем случае различных подстрок γ и β, для которых следует сформулировать подобные правила, бесконечное количество, поэтому они создаются только для узких подклассов КСГ. СЛАЙД 6.

Алгоритм ВСА работает следующим образом (СЛАЙД 7). Начиная с терминальной СФ из листьев дерева, мы пытаемся найти так называемую **связку** (*handle*) – RHS продукции, которую нужно заменить на LHS этой продукции, чтобы получить новый узел дерева разбора. Формально, связкой СФ μ называется такая левая подстрока β (возможно, пустая), что существует порождение $S \Rightarrow^* \alpha A\gamma \Rightarrow \alpha\beta\gamma \Rightarrow \mu$. С точки зрения процесса порождения алгоритм ВСА восстанавливает его, двигаясь от результирующей строки к аксиоме. Основной повторяющийся вопрос здесь такой. Какую из подстрок в промежуточной СФ выбрать в качестве связки, и каким нетерминалом ее заменить, чтобы получить предыдущую СФ, участвующую в порождении исходной терминальной строки и в итоге свернуть всю строку к аксиоме?

На СЛАЙДЕ 7 как раз и показаны несколько шагов простого переборного алгоритма, пытающегося решить эту задачу. На первом шаге в исходной терминальной строке *abbccabc* находим слева связку, подстроку *ab*, которая представляет собой RHS одной из продукций (*A -> ab*). После замены связки нетерминалом *A* получаем *Abccabc*. Через несколько шагов мы придем к тупиковой ситуации – СФ *ABAB*, в которой не может быть найдено ни одной связки, и дальнейшие свертки невозможны. Значит, потребуется откат. Чтобы его избежать и без возвратов строить дерево, для грамматики должен быть сформулирован критерий однозначного определения связки СФ порождаемого языка. Возможных СФ в общем случае бесконечное количество, поэтому найти такой критерий непросто.

К сожалению, алгоритм грубой силы имеет полиномиальную сложность. Хотелось бы получить в свое распоряжение более эффективные методы, которые также могут быть применены ко всем КСГ. В предыдущих разделах мы приводили примеры неоднозначных грамматик, которые приводят к нескольким деревьям разбора анализируемой строки.

Универсальные алгоритмы СА должны для любой данной КСГ G и входной строки α восстанавливать все возможные порождения строки α из G. Собственно говоря, именно этим и объясняется сложность универсальных алгоритмов.

Универсальные алгоритмы также делятся на НСА и ВСА. Мы рассмотрим два алгоритма СА для КСЯ: нисходящий – алгоритм Эрли и восходящий – алгоритм Кока-Янгера-Касами (он же – **CYK-алгоритм**). Оба этих алгоритма имеют вычислительную сложность $O(n^3)$.

Впоследствии будут продемонстрированы неуниверсальные (т.е. с потерей общности) алгоритмы СА линейной сложности, т.е. $O(n)$.

9.3 Алгоритм Эрли

Этот алгоритм является классическим алгоритмом НСА, который может быть применен ко всем КСГ, в том числе неоднозначным. Этот алгоритм является одним из эффективных общих алгоритмов. Мы дадим его не полностью формальное описание.

Алгоритм Эрли (АЭ) пытается построить все возможные нетерминалы из подстрок входной строки. Читая входную строку символ за символом, алгоритм для каждой

позиции во входной строке формирует список всех тех частично завершенных продукций грамматики, из которых прочтенный префикс входной строки и его части могут быть выведены, и после чтения очередного символа этот список модифицируется на основе полученной информации.

Входная строка $\alpha = a_1 \dots a_n$ читается слева направо. Для каждого входного символа a_i строится множество ситуаций M_i, определяющее состояние распознавателя после **анализа** этого символа. Каждая **ситуация** АЭ – это (СЛАЙД 8):

- Помеченная продукция *Pr*, согласно которой в данный момент считывается подстрока (или **связка**) входной строки, выводящаяся в соответствии с продукцией *Pr*.

- Место в продукции *Pr*, показывающее, какая доля RHS этой продукции уже распознана. Это место отмечается знаком '•' (Этот же символ мы будем использовать и для некоторых других алгоритмов СА).

- Указатель позиции во входной строке, после которой начался поиск возможности применения этой продукции.

В общем случае ситуацию мы будем записывать в формате $<A->\alpha\bullet\beta; \gamma>$, где $\alpha\beta$ – две возможно пустые части RHS, γ – правый контекст.

Будем считать, что исходная КСГ с аксиомой *S* дополнена продукцией *S*' -> #*S*#, где *S*' – это новая аксиома, а # – это псевдоскобки, в которые будет помещаться каждая терминальная строка, порождаемая исходной КСГ. Тогда начальное множество ситуаций (M_0 до анализа первого входного символа) будет содержать единственную ситуацию <*S*' -> •#*S*#; 0>. Метка стоит перед началом единственной RHS, которой может быть заменен *S*', а указатель равен 0. Именно после 0-го символа начинается подстрока (в нашем случае – вся анализируемая строка), для которой ищется порождение по продукции *S*'->#*S*#. СЛАЙД 9.

Множество ситуаций изменяется следующими операторами, соответственно, предсказания («Предсказатель»), считывания («Считыватель») и завершения («Завершатель»).

- **Предсказатель**. Если в M_i есть ситуация $<A->\alpha\bullet B\beta; q>$, то во множество добавляется ситуация $<B->\bullet\gamma, i>$ для всех продукций вида $B->\gamma$, имеющих *B* в LHS. Цель и смысл – следующие. После *i*-го символа входной строки, начиная с a_{i+1}, распознаватель ищет любую подстроку, которую можно породить из *B*. Назовем ситуацию $<A->\alpha\bullet B\beta; q>$ – **родительской**, $<B->\bullet\gamma ; i>$ – **порожденной**. СЛАЙД 10.

- **Считыватель**. Если в M_i есть ситуация $<A->\alpha\bullet b\beta; q>$, и если *b* – очередной терминал a_{i+1} входной строки, то в следующее множество ситуаций M_{i+1} добавляется ситуация $<A->\alpha\bullet b\beta; q>$. СЛАЙД 11.

- **Завершатель**. Он применяется к любой ситуации вида $<A->\alpha\bullet; q>$ в множестве M_i. Завершение продукции показывает, что она успешно применена, и из нетерминала *A*, начиная с шага *q*, порождена строка, совпадающая с $a_q a_{q+1} \dots a_i$ согласно продукции $A->\alpha$. Формально, $A => \alpha =>^* a_q a_{q+1} \dots a_i$. По этой причине в множестве M_q завершатель ищет ситуацию $<A->\bullet\alpha; q>$, и для каждой ситуации $<B->\gamma A\bullet\mu; s>$, которая является родительской для $<A->\bullet\alpha; q>$ в M_i он добавляет ситуацию $<B->\gamma A\bullet\mu; s>$. Это свидетельство того, что во входной строке подстрока $a_s a_{s+1} \dots a_i$ может быть порождена из начальной части продукции для нетерминала *B*, а именно $B => \gamma A\mu =>^* a_s a_{s+1} \dots a_i\mu$. СЛАЙД 12.

С каждым множеством ситуаций M_i предсказатель, считыватель и завершатель работают до тех пор, пока в M_i и M_{i+1} не перестанут появляться новые ситуации. Входная строка принимается, если в заключительном множестве ситуаций, т.е. после последнего символа входной цепочки, встречается ситуация <*S*'->#*S*#•; 0>. СЛАЙД 13.

Пример 63.

Пусть дана грамматика с продукциями *S*' -> #*E*#, *E*->*E*+*T* | *T*, *T* -> *T***P* | *P*, *P* -> *a*.

Построим множества ситуаций для АЭ на примере анализа строки # *a*+*a* #. См.

СЛАЙД 14.

Рассмотрим, как строятся множества ситуаций.

Начинаем с M_0. Оно, как указывалось выше, включает только ситуацию <*S*' -> •#*E*#; 0>, которая говорит о том, что до прихода первого символа в нулевой позиции мы ожидаем строку, порожденную из строки #*E*# из аксиомы *S*'. Применим к M_0 все операции, которые выполняются предсказателем, считывателем и завершателем. Считыватель добавляет в следующее множество (M_1) ситуацию <*S*' -> #•*E*#; 0>. Больше ни одного из операторов применить к M_0 нельзя.

Продолжаем с M_1. Это множество изначально включает ситуацию <*S*' -> #•*E*#; 0>, полученную из M_0. Предсказатель добавит еще пять ситуаций следующим образом. Поскольку в ситуации <*S*' -> #•*E*#; 0> указатель находится перед нетерминалом *E*, сюда добавляются две ситуации <*E* -> •*E*+*T*; 1> и <*E* -> •*T*; 1>, которые говорят о том, что с первой позиции мы ожидаем строку, порожденную из *E* по продукции *E->E+T* либо по продукции *E->T*. Остальные три ситуации добавляются повторным применением предсказателя. Считыватель добавляет в следующее множество (M_2) ситуацию <*P* ->*a*•; 1>. На этом построение множества M_1 заканчивается.

Продолжаем с M_2. Это множество изначально включает ситуацию <*P* ->*a*•; 1>, полученную из M_1 считывателем со сдвигом указателя положения за встреченный терминал *a*, т.к. он является очередным входным символом. Завершатель заставляет добавить в множество M_2 ситуацию <*T* ->*P*•; 1>. Дальнейшее применение завершателя вынуждает добавить в M_2 еще четыре ситуации <*E* ->*T*•; 1>, <*T* ->*T*•**P*; 1>, <*E* ->*E*•+*T*; 1> и <*S*' ->#*E*•#; 0>. Ни в одной из ситуаций в M_2 указатель положения не стоит перед нетерминалом, поэтому предсказателю здесь делать нечего. Считыватель добавляет в следующее множество (M_3) ситуацию <*E* ->*E*+•*T*; 1>. На этом построение множества M_2 заканчивается.

Множества M_3 – M_5 строятся аналогично.

Из схемы видно, что все ситуации связаны указателями. Указатель от ситуации K_i к ситуации K_j проводятся тогда, когда K_j порождена из K_i любым из трех операторов.

Кроме того, видно, что часть ситуаций в построенных множествах несущественные. Например, в M_4 это ситуации <*E* ->*E*•+*T*; 1> и <*T* ->•*T***P*; 3>, которые не приводят к завершающим ситуациям вида <*A*->α•; *k*>. Последовательное выбрасывание таких ситуаций приводят к связному списку существенных ситуаций. См. СЛАЙД 15.

Восстановление дерева разбора входной строки выполняется в АЭ после построения всех множеств ситуаций, если последнее множество содержит <*S*'->#*E*#•; 0>. Дерево легко восстанавливается по списку существенных ситуаций. На СЛАЙДЕ 16 показано такое дерево из примера 63. На СЛАЙДЕ 15 более ясным образом представлены зависимости ситуаций. Горизонтальные указатели связывают одну и ту же продукцию с последовательно перемещаемой «меткой». Такие ситуации последовательно порождаются считывателем и завершателем. Вертикальные указатели связывают ситуации, порожденные предсказателем. Для неоднозначных строк из одного нетерминала может идти более чем один вертикальный указатель, что демонстрирует возможность неоднозначного построения дерева разбора.

По схеме на СЛАЙДЕ 16 очевидна справедливость следующей теоремы, которую мы принимаем без доказательства. СЛАЙД 17.

Теорема 9.1. Ситуация <*A*->α•β; *i*> находится во множестве M_j тогда и только тогда, когда существует такое порождение S=>*$\gamma A\delta$, что:

1. γ =>* $a_1 \dots a_i$
2. α =>* $a_{i+1} \dots a_j$

Еще раз отметим следующие особенности АЭ. Во-первых, он универсальный, т.е. подходит для всех КСГ.

Во-вторых, он выполняет нисходящий анализ и восстанавливает левое порождение.

В-третьих, он строит состояния распознавателя динамически, непосредственно при обработке конкретной строки языка. Для каждой такой строки множества ситуаций, строящиеся при ее распознавании, уникальны. Каждой такое множество связано с конкретным местом в анализируемой строке.

В общем случае, АЭ затрачивает $O(n^3)$ шагов при распознавании строки длиной n, используя емкость памяти $O(n^2)$. Если же КСГ однозначна и приведена в НФХ, то алгоритму понадобится $O(n^2)$ шагов. Автор алгоритма утверждал, что для широкого класса однозначных грамматик время распознавания линейно.

Мы описали бесконтекстную версию алгоритма, однако автор алгоритма рекомендует добавить в каждую ситуацию терминальную строку длиной k – **допустимый терминальный контекст**, который может встретиться в строках языка после того, как данная продукция была применена. Контекст используется завершателем для принятия решения о возможности применения закончившейся продукции. Приписанный ей контекст сравнивается с последующей подстрокой длины k, и продукция применятся только тогда, когда эти строки совпадают. СЛАЙД 18.

Использование контекста может повысить эффективность распознавателя, т.к. лишние ветви вычислений могут быть отброшены раньше, и на их обработку время и память не будут тратиться. С другой стороны, на ведение и расчет контекста тоже требуются ресурсы. Автора алгоритма не рекомендовал использовать контекст длины более единицы.

9.4 Алгоритм Кока-Янгера-Касами

Этот алгоритм восстанавливает снизу вверх все возможные порождения сначала для всех подстрок входной строки длиной 1, затем – длиной 2 и т.д. до тех пор, пока не будет проанализирована вся строка. Основная идея алгоритма заключается в том, что результаты СА более коротких подстрок, полученные на предыдущих шагах, используются для определения тех продукций, которые могут быть использованы для порождения более длинных строк. КСГ в этом случае должна быть приведена в НФХ, что позволяет анализировать разбиение любой подстроки входной строки длиной более 1 на две части, СА каждой из которых уже произведен на предыдущих шагах и запомнен в таблице. СЛАЙД 19.

Это универсальный алгоритм для всех КСГ, он осуществляет восходящий СА, затрачивая $O(n^3)$ шагов при распознавании строки длиной n, используя емкость памяти $O(n^2)$. Был независимо с небольшими вариациями получен тремя учеными Дж.Коком, Д.Янгером и Т.Касами, далее мы будем использовать анонсированное выше название – CYK-алгоритм.

В процессе работы CYK-алгоритм строит треугольную **таблицу разбора** (таблицу синтаксического анализа, далее – ТСА) T непосредственно по анализируемой последовательности символов. В каждую ячейку t_{ik} ТСА помещается множество нетерминалов, из которых можно вывести отрезок выходной строки длиной k, начиная с a_i: $t_{ij} = \{A \mid A =>^* a_i \dots a_{i+j+1}\}$. Содержимое ячейки ТСА вычисляется очень просто по КСГ в НФХ. Действительно, $t_{i1} = \{A | A \to a_i \in P\}$, а $t_{i1} = \{A | A \to BC \in P \,\&\, (\exists k : 1 \le k < j) : B \in t_{ik} \,\&\, C \in t_{i+k,j-k}\}$. Входная строка принадлежит языку, порождаемому грамматикой, если в ячейке t_{1n} есть аксиома. СЛАЙД 20.

ТСА заполняется снизу вверх, причем каждая строка соответствует определенной длине подстрок: Нижняя – подстрокам длины 1, вторая снизу – длины 2 и так далее вплоть до верхней строки, соответствующей одной подстроке длины n, т.е. всей анализируемой строки.

Нижняя строка ТСА T заполняется так: в t_{i1} помещаются все нетерминалы A, для которых есть продукция $A \to a_i$. Предположим, уже заполнены все строки таблицы T от 1

до j-1 включительно. Рассмотрим ячейку t_{ij}. Она соответствует фрагменту $a_i \ldots a_{i+j-1}$. Этот фрагмент разбивается всеми возможными способами на пары соседних строк $<a_i>$ и $<a_{i+1} \ldots a_{i+j-1}>$ и $<a_{i+2}>$ и $<a_{i+2} \ldots a_{i+j-1}>$ и т.д. Каждому варианту разбиения соответствует пара ячеек ТСА, в которых стоят нетерминалы, из которых могут быть порождены соответствующие строки. Пусть эта пара ячеек – (t', t''). В рассматриваемую ячейку t_{ij} помещается нетерминал A, если среди продукций есть A -> BC, и нетерминал B входит в ячейку t', а C – в t''. СЛАЙД 21.

Рассмотрим, к примеру, ячейку t_{34} из ТСА, представленной на СЛАЙДЕ 22, для строки из шести символов. В нее должны быть помещены нетерминалы, из которых выводится фрагмент выходной строки, начинающийся с a_3, длина фрагмента – четыре символа. Очевидно, это последовательность символов $a_3a_4a_5a_6$. Эта подстрока может быть разбита на пары непустых соседних фрагментов: (1) $<a_3>$ и $<a_4a_5a_6>$; (2) $<a_3a_4>$ и $<a_5a_6>$; (3) $<a_3a_4a_5>$ и $<a_6>$. Этим трем парам соответствуют пары ячеек, где могут стоять нетерминалы, из которых эти подстроки порождаются. Для пары (1) это t_{31} и t_{43}; для пары (2) это t_{32} и t_{52}; для пары (3) это t_{33} и t_{61}. Если в первой ячейке пары есть нетерминал B, во второй – нетерминал C, а среди продукций есть A -> BC, то нетерминал A помещается в ячейку t_{34}. Это схематично показано на СЛАЙДЕ 22.

Пример 64.

Приведем СА строки ()()() из языка скобок с использованием неоднозначной грамматики с продукциями S -> SS | LR, L -> (,. R ->). ТСА представлена на СЛАЙДЕ 23. В ней нетерминалы, стоящие в ячейке t_{ij}, помечены индексом ij.

Второй шаг алгоритма – это восстановление дерева разбора строки. Оно осуществляется с помощью рекурсивной процедуры $REDUCE(i, j, A)$, которая дает левое порождение $A \Rightarrow^* a_ia_{i+1} \ldots a_{j-1}$.

Эту процедуру легко построить:

1. Если $j = 1$ и A -> a_i – это продукция с номером m, то выдать m.

2. Пусть $j > 1$. Выберем k – наименьшее из чисел, для которых существуют B из t_{ik}, C из $t_{i+k,j-k}$ и продукция A -> BC с номером m. Выберем эту продукцию, выдаем m и выполняем $REDUCE(i, k, B)$ и $REDUCE(i+1, j-k, C)$.

Работа процедуры начинается с $REDUCE(1, n, S)$. СЛАЙД 24.

На СЛАЙДЕ 25 для ячейки t_{16} показаны два возможных варианта включения нетерминала S_{16} в эту ячейку: либо по продукции S_{16} -> $S_{12}S_{34}$, либо по продукции S_{16} -> $S_{14}S_{34}$. Оба варианта возможны, поэтому мы можем построить два дерева разбора этой строки по заданной грамматике.

Литература к лекции 9

1. Контекстно-свободная грамматика - http://ru.wikipedia.org/wiki/Контекстно-свободная_грамматика
2. Серебряков В. А., Галочкин М. П., Гончар Д. Р., Фуругян М. Г. Теория и реализация языков программирования — М.: МЗ-Пресс, 2006 г., 2-е изд. - http://trpl7.ru/t-books/TRYAP_BOOK_Details.htm
3. Контекстно-свободные грамматики - http://www.math.spbu.ru/user/mbk/PDF/Ch-4.pdf
4. J. Earley, "An efficient context-free parsing algorithm", Communications of the Association for Computing Machinery, 13:2:94-102, 1970.
5. Earley parser - http://en.wikipedia.org/wiki/Earley_parser
6. CYK algorithm - http://en.wikipedia.org/wiki/CYK_algorithm
7. John Cocke and Jacob T. Schwartz (1970). Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University.
8. T. Kasami (1965). An efficient recognition and syntax-analysis algorithm for context-free languages. Scientific report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford,

MA.
9. Daniel H. Younger (1967). Recognition and parsing of context-free languages in time n3. Information and Control 10(2): 189–208.