МОДУЛЬ 2

Раздел 3. Регулярные грамматики

Рассмотрим методы и средства, которые обычно используются при построении лексических анализаторов. В основе таких анализаторов лежат регулярные грамматики, поэтому рассмотрим грамматики этого класса более подробно.

Соглашение: в дальнейшем, если особо не оговорено, под **регулярной грамматикой будем понимать леволинейную автоматную грамматику** $G = \langle T, N, P, S \rangle$ без пустых правых частей. Напомним, что в такой грамматике каждое правило из P имеет вид $A \to Bt$ либо $A \to t$, где $A, B \in N$, $t \in T$.

Соглашение: предположим, что анализируемая цепочка заканчивается специальным символом $\perp$ - *признаком конца цепочки*.

Для грамматик этого типа существует алгоритм определения того, принадлежит ли анализируемая цепочка языку, порождаемому этой грамматикой (*алгоритм разбора*).

3.1 Разбор по регулярным грамматикам

Алгоритм разбора леволинейной автоматной грамматики:

(1) первый символ исходной цепочки $a_1a_2...a_n\perp$ заменяем нетерминалом A, для которого в грамматике есть правило вывода $A \to a_1$ (другими словами, производим свертку терминала a_1 к нетерминалу A)

(2) затем многократно (до тех пор, пока не считаем признак конца цепочки) выполняем следующие шаги: полученный на предыдущем шаге нетерминал A и расположенный непосредственно справа от него очередной терминал a_i исходной цепочки заменяем нетерминалом B, для которого в грамматике есть правило вывода $B \to Aa_i$ ($i = 2, 3, ..., n$);

Это эквивалентно **построению дерева разбора методом снизу-вверх**: на каждом шаге алгоритма строим один из уровней в дереве разбора, поднимаясь от листьев к корню.

(3) первый символ исходной цепочки $a_1a_2...a_n\perp$ заменяем нетерминалом A, для которого в грамматике есть правило вывода $A \rightarrow a_1$ (другими словами, производим свертку терминала a_1 к нетерминалу A)

(4) затем многократно (до тех пор, пока не считаем признак конца цепочки) выполняем следующие шаги: полученный на предыдущем шаге нетерминал A и расположенный непосредственно справа от него очередной терминал a_i исходной цепочки заменяем нетерминалом B, для которого в грамматике есть правило вывода $B \rightarrow Aa_i$ (i = 2, 3, ..., n);

Это эквивалентно построению дерева разбора методом снизу-вверх: на каждом шаге алгоритма строим один из уровней в дереве разбора, поднимаясь от листьев к корню.

При работе этого алгоритма возможны следующие ситуации:

(1) прочитана вся цепочка; на каждом шаге находилась единственная нужная свертка; на последнем шаге свертка произошла к символу S. Это означает, что исходная цепочка $a_1a_2...a_n\perp \in L(G)$.

(2) прочитана вся цепочка; на каждом шаге находилась единственная нужная свертка; на последнем шаге свертка произошла к символу, отличному от S. Это означает, что исходная цепочка $a_1a_2...a_n\perp \notin L(G)$.

(3) на некотором шаге не нашлось нужной свертки, т. е. для полученного на предыдущем шаге нетерминала A и расположенного непосредственно справа от него очередного терминала a_i исходной цепочки не нашлось нетерминала B, для которого в грамматике было бы правило вывода $B \rightarrow Aa_i$. Это означает, что исходная цепочка $a_1a_2...a_n\perp \notin L(G)$.

на некотором шаге работы алгоритма оказалось, что есть более одной подходящей свертки, т. е. в грамматике разные нетерминалы имеют правила вывода с одинаковыми правыми частями, и поэтому непонятно, к какому из них производить свертку. Это говорит о ***недетерминированности разбора***.

Допустим, что разбор на каждом шаге детерминированный.

Лексический анализ можно выполнить на основе таблицы не терминальных и терминальных символов или диаграммы состояний

Для того, чтобы быстрее находить правило с подходящей правой частью, зафиксируем все возможные свертки (это определяется только грамматикой и не зависит от вида анализируемой цепочки).

3.2 Таблица лексического анализа

Создадим таблицу, столбцы которой помечены терминальными символами. Таблица фиксирует все возможные свертки.

Первая строка помечена символом H ($H \notin N$), а значение каждого элемента этой строки - это нетерминал, к которому можно свернуть помечающий столбец терминальный символ.

Остальные строки помечены нетерминальными символами грамматики. Значение каждого элемента таблицы, начиная со второй строки — это нетерминальный символ, к которому можно свернуть пару «нетерминал-терминал», которыми помечены соответствующие строка и столбец.

Например, для леволинейной грамматики $G_{left} = \langle \{a, b, \perp\}, \{S, A, B, C\}, P, S \rangle$, где P:

$S \rightarrow C\perp$

$C \rightarrow Ab \mid Ba$

$A \rightarrow a \mid Ca$

$B \rightarrow b \mid Cb,$

такая таблица будет выглядеть следующим образом:

	a	***b***	⊥
H	*A*	*B*	—
C	*A*	*B*	*S*
A	—	*C*	—
B	*C*	—	—
S	—	—	—

Знак «—» ставится в том случае, если соответствующей свертки нет.

Но чаще информацию о возможных свертках представляют в виде *диаграммы состояний (ДС)* - неупорядоченного ориентированного помеченного графа. Граф строится следующим образом.

3.3 Диаграмма состояний (ДС) лексического анализа

Диаграмма состояний (ДС) - неупорядоченный ориентированный помеченный граф.

Граф строится следующим образом:

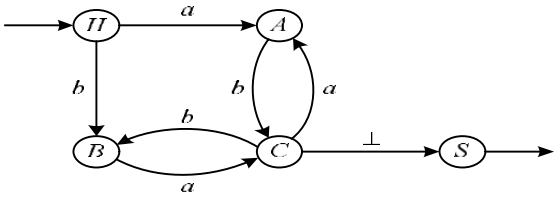
(1) строим вершины графа, помеченные нетерминалами грамматики (для каждого нетерминала - одну вершину), и еще одну вершину, помеченную символом, отличным от нетерминальных, например, *H*. Эти вершины будем называть *состояниями*. *H* — начальное состояние.

(2) соединяем эти состояния дугами по следующим правилам:

а) для каждого правила грамматики вида $W \to t$ соединяем дугой состояния H и W (от H к W) и помечаем дугу символом t;

б) для каждого правила $W \to Vt$ соединяем дугой состояния V и W (от V к W) и помечаем дугу символом t.

Диаграмма состояний для грамматики G_{left}



Алгоритм разбора по диаграмме состояний

(1) объявляем текущим начальное состояние H;

(2) затем многократно (до тех пор, пока не считаем признак конца цепочки) выполняем следующие шаги: считываем очередной символ исходной цепочки и переходим из текущего состояния в другое состояние по дуге, помеченной этим символом. Состояние, в которое мы при этом попадаем, становится текущим.

При работе этого алгоритма возможны следующие ситуации (аналогичные ситуациям, которые возникают при разборе непосредственно по регулярной грамматике):

1) Прочитана вся цепочка; на каждом шаге находилась единственная дуга, помеченная очередным символом анализируемой цепочки; в результате последнего перехода оказались в состоянии S. Это означает, что исходная цепочка принадлежит $L(G)$.

2) Прочитана вся цепочка; на каждом шаге находилась единственная «нужная» дуга; в результате последнего шага оказались в состоянии, отличном от S. Это означает, что исходная цепочка не принадлежит $L(G)$.

3) На некотором шаге не нашлось дуги, выходящей из текущего состояния и помеченной очередным анализируемым символом. Это означает, что исходная цепочка не принадлежит $L(G)$.

На некотором шаге работы алгоритма оказалось, что есть несколько дуг, выходящих из текущего состояния, помеченных очередным анализируемым символом, но ведущих в разные состояния. Это говорит о *недетерминированности разбора*.

Диаграмма состояний определяет **конечный автомат**, построенный **по регулярной грамматике**, который допускает множество цепочек, составляющих язык, определяемый этой грамматикой.

3.4 Детерминированный конечный автомат (ДКА)

Детерминированный конечный автомат **(*ДКА*)** - это пятерка ⟨ ***K,T,*** **δ,** ***H, S*** ⟩, где

K — конечное множество состояний;

T — конечное множество допустимых входных символов;

δ — отображение множества $K \times T$ в K, определяющее поведение автомата;

H $\in K$ — начальное состояние;

S $\in K$ — заключительное состояние (либо множество заключительных состояний $S \subseteq K$).

Замечания к определению ДКА:

А) Заключительных состояний в ДКА может быть более одного, однако для любого регулярного языка, все цепочки которого заканчиваются маркером конца (⊥), существует ДКА с единственным заключительным состоянием. Заметим также, что ДКА, построенный по регулярной грамматике рассмотренным выше способом, всегда будет иметь единственное заключительное состояние S.

В) Отображение $\delta: K \times T \rightarrow K$ называют ***функцией переходов*** **ДКА**. $\delta(A, t) = B$ означает, что из состояния A по входному символу t происходит переход в состояние B. Иногда δ определяют лишь на подмножестве $K \times T$ (частичная функция). Если значение $\delta(A, t)$ не определено, то автомат не может дальше продолжать работу и останавливается в состоянии «ошибка».

Диаграмма состояний определяет **конечный автомат**, построенный **по регулярной грамматике**, который допускает множество цепочек, составляющих язык, определяемый этой грамматикой.

Состояния и дуги ДС — это графическое изображение функции переходов конечного автомата из состояния в состояние при условии, что очередной анализируемый символ совпадает с символом-меткой дуги.

Среди всех состояний выделяется начальное (считается, что в начальный момент своей работы автомат находится в этом состоянии) и **заключительное** (если автомат завершает работу переходом в это состояние, то анализируемая цепочка им допускается). На ДС эти состояния отмечаются

короткими входящей и соответственно исходящей стрелками, не соединенными с другими вершинами.

Определение: ДКА *допускает цепочку* $a_1a_2...a_n$, если

$\delta(H, a_1) = A_1$; $\delta(A_1, a_2) = A_2$; …;

$\delta(A_{n-2}, a_{n-1}) = A_{n-1}$; $\delta(A_{n-1}, a_n) = S$,

где $a_i \in T$, $A_j \in K$, $j = 1, 2, ..., n - 1$; $i = 1, 2, ..., n$. H - начальное состояние, S - заключительное состояние.

Определение: множество цепочек, допускаемых ДКА, составляет определяемый им ***язык***.

Для более удобной работы с диаграммами состояний введем несколько соглашений:

- если из одного состояния в другое выходит несколько дуг, помеченных разными символами, то будем изображать одну дугу, помечая ее списком из всех таких символов;
- непомеченная дуга будет соответствовать переходу при любом символе, кроме тех, которыми помечены другие дуги, выходящие из этого состояния;
- введем состояние ошибки (ER); переход в это состояние будет означать, что исходная цепочка языку не принадлежит.

По диаграмме состояний легко написать анализатор для регулярной грамматики.

Рассмотрим алгоритм анализа для праволинейной грамматики

$$G_{right} = \langle \{a, b, \perp\}, \{H, A, B, C\}, P, H \rangle$$

на примере цепочки $abba\perp$.

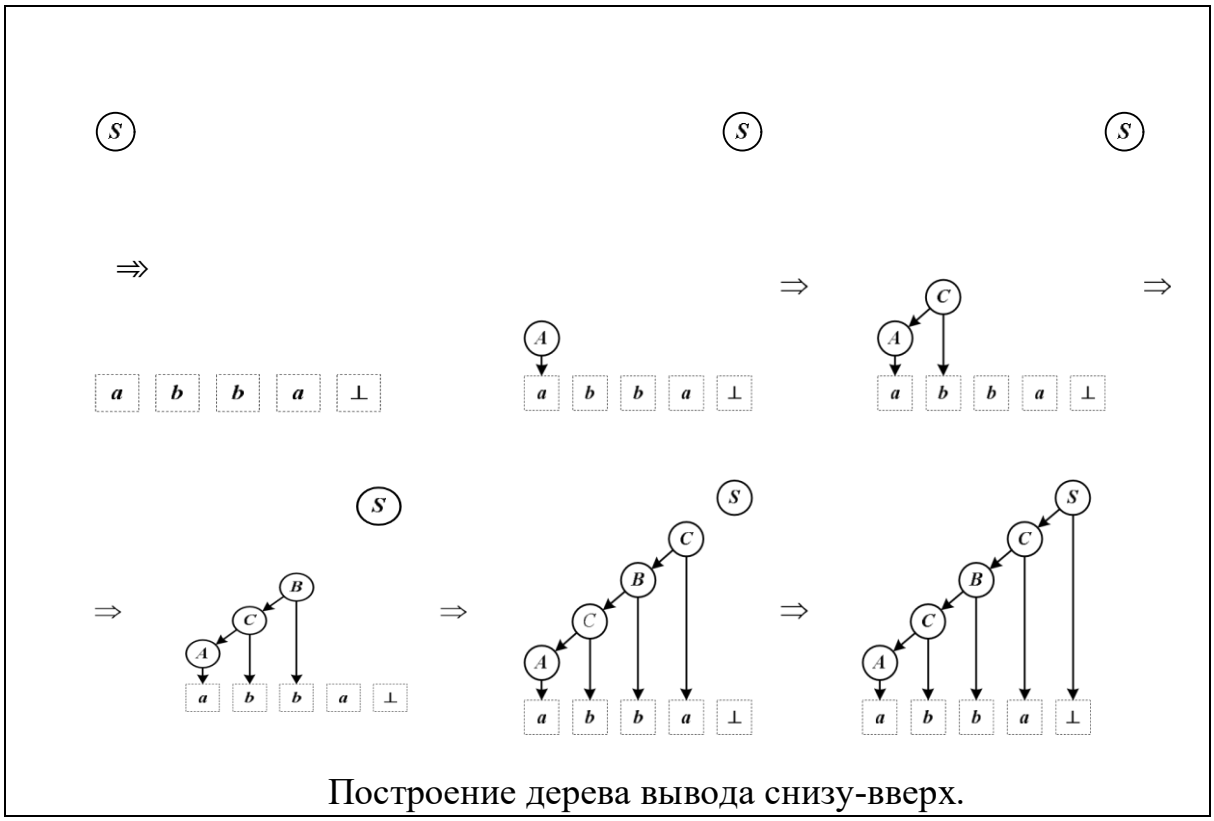
При анализе данной цепочки получим следующую последовательность переходов в ДС:

$$H \xrightarrow{a} A \xrightarrow{b} C \xrightarrow{b} B \xrightarrow{a} C \xrightarrow{\perp} S$$

Вспомним, что каждый переход в ДС означает свертку сентенциальной формы путем замены в ней пары «нетерминал-терминал» Nt на нетерминал L, где $L \rightarrow Nt$ правило вывода в грамматике. Такое применение правила в обратную сторону будем записывать с помощью обратной стрелки $Nt \leftarrow L$ (обращение правила вывода). Тогда получим следующую последовательность сверток, соответствующую переходам в ДС:

$$abba\perp \leftarrow \boldsymbol{A}bba\perp \leftarrow \boldsymbol{C}ba\perp \leftarrow \boldsymbol{B}a\perp \leftarrow \boldsymbol{C}\perp \leftarrow \boldsymbol{S}$$

Эта последовательность не что иное, как обращение (правого) вывода цепочки $abba\perp$ в грамматике G. Она соответствует построению дерева снизу-вверх.



Построение дерева вывода снизу-вверх.

Разбор по праволинейной грамматике.

По диаграмме состояний построим праволинейную автоматную грамматику G_{right} следующим способом:

- нетерминалами будут состояния из ДС (кроме S);
- каждой дуге из состояния V в заключительное состояние S (помеченной признаком конца $\perp$) будет соответствовать правило $V \to \perp$;
- каждой дуге из состояния V в состояние W, помеченной символом t, будет соответствовать правило $V \to tW$;
- начальное состояние H объявляется начальным символом грамматики.

$$G_{right} = \langle \{a, b, \perp\}, \{H, A, B, C\}, P, H \rangle$$

P:

$H \to aA \mid bB$

$A \to bC$

$C \to bB \mid aA \mid \perp$

$B \to aC$

Заметим, что $L(G_{right}) = L(G_{left})$, так как грамматики G_{right} и G_{left} соответствуют одной и той же ДС

Нетрудно описать и обратный алгоритм для праволинейной автоматной грамматики, если все ее правила с односимвольной правой частью имеют вид $V \to \perp$. Состояниями ДС будут нетерминалы грамматики и еще одно специальное заключительное состояние S, в которое для каждого правила вида $V \to \perp$ проводится дуга из V, помеченная признаком конца $\perp$. Для каждого правила вида V → t W проводится дуга из V в W, помеченная символом t. Начальным состоянием в ДС будет начальный символ H.

3.5 О недетерминированном разборе

Определение.

Недетерминированный конечный автомат (НКА) — это пятерка $\langle K, T, \delta, H, S \rangle$, где

K — конечное множество состояний;

T — конечное множество допустимых входных символов;

δ — отображение множества $K \times T$ в множество подмножеств K;

$H \subseteq K$ — конечное множество начальных состояний;

$S \subseteq K$ — конечное множество заключительных состояний.

$\delta\ (A,\ t) = \{B_1,\ B_2,...,\ B_n\}$ означает, что из состояния A по входному символу t можно осуществить переход в любое из состояний B_i, $i = 1, 2, ..., n$.

В алгоритме разбора по ДС для леволинейной грамматике было отмечено, что несколько нетерминалов могут иметь одинаковые правые части, и поэтому неясно, к какому из них делать свертку.

При анализе по праволинейной грамматике может оказаться, что нетерминал имеет две правые части с одинаковыми терминальными символами и поэтому неясно, на какую альтернативу заменить нетерминал.

В терминах диаграммы состояний эти ситуации означают, что из одного состояния выходит несколько дуг, ведущих в разные состояния, но помеченных одним и тем же символом.

В алгоритме разбора по ДС для леволинейной грамматике было отмечено, что несколько нетерминалов могут иметь одинаковые правые части, и поэтому неясно, к какому из них делать свертку. (см. ситуацию 4 в описании алгоритма).

Для грамматики $G = \langle \{a, b, \perp\}, \{S, A, B\}, P, S \rangle$, где

P:

$S \to A\perp$

$A \to a \mid Bb$

$B \to b \mid Bb$

разбор будет недетерминированным (т.к. у нетерминалов A и B есть одинаковые правые части — Bb).

Такой грамматике будет соответствовать недетерминированный конечный автомат.

Любой НКА, также как и ДКА, можно представить в виде таблицы (в одной ячейке такой таблицы можно указывать сразу несколько состояний, в которые возможен переход из заданного состояния по текущему символу) или в виде диаграммы состояний (ДС).

В ДС каждому состоянию из множества K соответствует вершина; из вершины A в вершину B ведет дуга, помеченная символом t, если $B \in \boldsymbol{\delta}(A, t)$. (В НКА из одной вершины могут исходить несколько дуг с одинаковой пометкой). ***Успешный путь*** - это путь из начальной вершины в заключительную; ***пометка пути*** - это последовательность пометок его дуг. *Язык,* допускаемый НКА, - это множество пометок всех успешных путей.

Если начальное состояние автомата (НКА или ДКА) одновременно является и заключительным, то автомат допускает пустую цепочку ε.

Замечание

Автомат, построенный по регулярной грамматике без пустых правых частей, не допускает ε.

Для построения разбора по регулярной грамматике в недетерминированном случае можно предложить алгоритм, который будет перебирать все возможные варианты сверток (переходов) один за другим; если цепочка принадлежит языку, то будет найден успешный путь; если каждый из просмотренных вариантов завершится неудачей, то цепочка языку не принадлежит. Однако такой алгоритм практически неприемлем, поскольку при переборе вариантов мы, скорее всего, снова окажемся перед проблемой выбора и, следовательно, будем иметь «дерево отложенных вариантов» и экспоненциальный рост сложности разбора.

Один из наиболее важных результатов теории конечных автоматов состоит в том, что класс языков, определяемых недетерминированными конечными автоматами, совпадает с классом языков, определяемых детерминированными конечными автоматами.

Утверждение. Пусть L - формальный язык. Следующие утверждения эквивалентны:

(1) L порождается регулярной грамматикой;

(2) L допускается ДКА;

(3) L допускается НКА.

Эквивалентность пунктов (1) и (2) следует из рассмотренных выше алгоритмов построения конечного автомата по регулярной грамматике и обратно — грамматики по автомату. Очевидно, что из (2) следует (3): достаточно записать вместо каждого перехода ДКА $\boldsymbol{\delta}(C, a) = b$ эквивалентный ему переход в НКА $\boldsymbol{\delta}(C, a) = \{b\}$, начальное состояние ДКА поместить в множество начальных состояний НКА, а заключительное состояние ДКА поместить в множество заключительных состояний НКА. Далее рассмотрим алгоритм построения ДКА, эквивалентного НКА, - он обосновывает то, что из (3) следует (2).

3.6 Алгоритм построения ДКА по НКА

Вход: НКА $M = \langle K, T, \boldsymbol{\delta}, H, S \rangle$.

Выход: ДКА $M_1 = \langle K_1, T, \boldsymbol{\delta}_1, H_1, S_1 \rangle$, допускающий тот же язык, что и автомат M: $L(M) = L(M_1)$.

Метод:

1. Элементами K_1, т. е. состояниями в ДКА, будут некоторые подмножества множества состояний НКА. Заметим, что в силу конечности множества K, множество K_1 также конечно и имеет не более 2^s элементов, где s — мощность K.

Подмножество $\{A_1, A_2, \ldots, A_n\}$ состояний из K будем для краткости записывать как $\underline{A_1A_2...A_n}$. Множество K_1 и переходы, определяющие функцию $\boldsymbol{\delta}_1$, будем строить, начиная с состояния H_1: $H_1 := \underline{A_1A_2...A_n}$, где $A_1, A_2, \ldots, A_n \in H$. Другими словами, все начальные состояния НКА M объединяются в одно состояние H_1 для ДКА M_1. Добавляем в множество K_1 построенное начальное состояние H_1 и пока считаем его нерассмотренным (на втором шаге оно рассматривается и строятся остальные состояния множества K_1, а также переходы $\boldsymbol{\delta}_1$.)

2. Пока в K_1 есть нерассмотренный элемент $\underline{A_1A_2...A_m}$, «рассматриваем» его и выполняем для каждого $t \in T$ следующие действия:

▪ Полагаем $\boldsymbol{\delta}_1$ ($\underline{A_1A_2...A_m}$, t) = $\underline{B_1B_2...B_k}$, где для $1 \le j \le k$ в НКА $\boldsymbol{\delta}$ (A_i, t) = B_j для некоторых $1 \le i \le m$. Другими словами, $\underline{B_1B_2...B_k}$ — это множество всех состояний в НКА, куда можно перейти по символу t из множества состояний $\underline{A_1A_2...A_m}$. В ДКА M_1 получается детерминированный переход по символу t из состояния $\underline{A_1A_2...A_m}$ в состояние $\underline{B_1B_2...B_k}$. (Если $k = 0$, то полагаем $\boldsymbol{\delta}_1$ ($\underline{A_1A_2...A_m}$, t) = $\emptyset$).

▪ Добавляем в K_1 новое состояние $\underline{B_1B_2...}$ $\underline{B_k}$.

Шаг 2 завершается, поскольку множество новых состояний K_1 конечно.

3. Заключительными состояниями построенного ДКА M_1 объявляются все состояния, содержащие в себе хотя бы одно заключительное состояние НКА M:

Пояснения к алгоритму построения ДКА по НКА

Множество S_1 построенного ДКА может состоять более, чем из одного элемента. Не для всех регулярных языков существует ДКА с единственным заключительным состоянием (пример: язык всех цепочек в алфавите $\{a, b\}$, содержащих не более двух символов b). Однако для реализации алгоритма детерминированного разбора заключительное состояние должно быть единственным. В таком случае изменяют входной язык, добавляя маркер $\perp$ в конец каждой цепочки (на практике в роли маркера конца цепочки $\perp$ может выступать признак конца файла, символ конца строки или другие

разделители). Вводится новое состояние S, и для каждого состояния Q из множества S_1 добавляется переход по символу $\perp$: $\boldsymbol{\delta}_1$ (Q,$\perp$) = S. Состояния из S_1 больше не считаются заключительными, а S объявляется единственным заключительным состоянием. Теперь по такому ДКА можно построить автоматную грамматику, допускающую детерминированный разбор.

Проиллюстрируем работу алгоритма на примерах.

Пример 1.

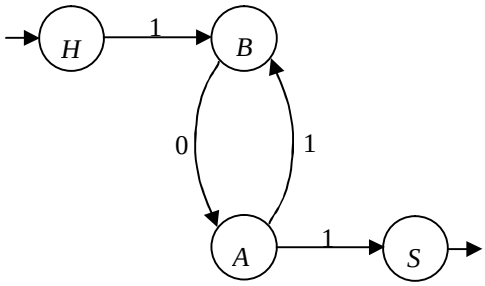
Пусть имеется НКА $M = \langle\{ H, A, B, S \}, \{0, 1\}, \boldsymbol{\delta}, \{H \}, \{S \} \rangle$, где

$\boldsymbol{\delta}$ (H, 1) = $\{B\}$

$\boldsymbol{\delta}$ (A, 1) = $\{B, S\}$

$\boldsymbol{\delta}$ (B, 0) = $\{A\}$

$L(M) = \{1(01)^n | \, n \geq 1\}$.



ДС для автомата M .

Грамматика, соответствующая M:

$S \quad \rightarrow A1$

$A \quad \rightarrow B0$

$B \quad \rightarrow A1 \mid 1$

Построим ДКА по НКА, пользуясь предложенным алгоритмом. Начальным состоянием будет $\underline{H}$.

$\boldsymbol{\delta}_1(\underline{H}, 1) = \underline{B}$

$\boldsymbol{\delta}_1(\underline{B}, 0) = \underline{A}$

$\boldsymbol{\delta}_1(\underline{A}, 1) = \underline{BS}$

$\boldsymbol{\delta}_1(\underline{BS}, 0) = \underline{A}$

Заключительным состоянием построенного ДКА является состояние $\underline{BS}$.

Таким образом, $M_1 = \langle\{\underline{H}, \underline{B}, \underline{A}, \underline{BS}\}, \{0, 1\}, \boldsymbol{\delta}_1, \underline{H}, \underline{BS} \rangle$. Для удобства переименуем состояния в M_1: $\underline{BS}$ обозначается теперь как S_1, а в однобуквенных именах состояний вместо подчеркивания используется индекс 1.

Тогда $M_1 = \langle\{H_1, B_1, A_1, S_1\}, \{0, 1\}, \{\boldsymbol{\delta}_1(H_1, 1) = B_1; \boldsymbol{\delta}_1 (B_1, 0) = A_1; \boldsymbol{\delta}_1 (A_1, 1) = S_1; \boldsymbol{\delta}_1 (S_1, 0) = A_1\}, H_1, S_1\rangle$.

Грамматика, соответствующая M_1:

$S_1 \rightarrow A_1 1$

$A_1 \rightarrow S_1 0 \mid B_1 0$

$B_1 \rightarrow 1$

Построим диаграмму состояний (рис. 5).

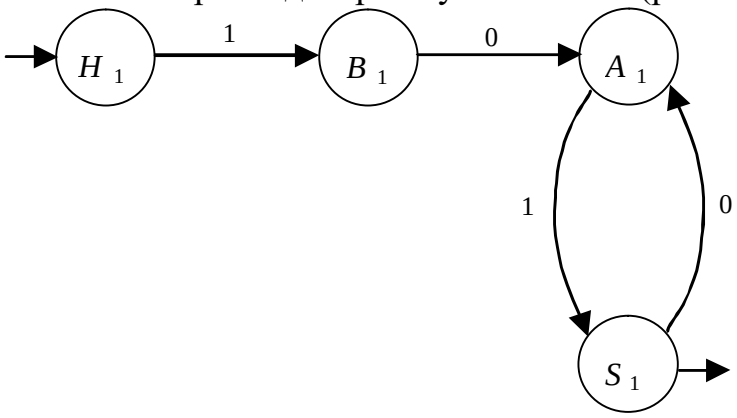


Рис. 5 . Диаграмма состояний M_1 .

Пример 2.

Дана регулярная грамматика $G = \langle T, N, P, S \rangle$

P:

$S \rightarrow Sb \mid Aa \mid a$

$A \rightarrow Aa \mid Sb \mid b$,

которой соответствует НКА. С помощью преобразования НКА в ДКА построить грамматику, по которой возможен детерминированный разбор.

Построим функцию переходов НКА:

$\boldsymbol{\delta}(H, a) = \{S\}$

$\boldsymbol{\delta}(H, b) = \{A\}$

$\boldsymbol{\delta}(A, a) = \{A, S\}$

$\boldsymbol{\delta}(S, b) = \{A, S\}$

Процесс построения функции переходов ДКА удобно изобразить в виде таблицы, начав ее заполнение с начального состояния $\underline{H}$, добавляя строки для вновь появляющихся состояний:

состояние \ символ	*a*	*b*
H	*S*	*A*
S	∅	*AS*
A	*AS*	∅
AS	*AS*	*AS*

появились состояния *S* и *A*, они рассматриваются в следующих строках таблицы

Переходы ДКА:

$\delta_1(H, a) = S$

$\delta_1(H, b) = A$

$\delta_1(S, b) = AS$

$\delta_1(A, a) = AS$

$\delta_1(AS, a) = AS$

$\delta_1(AS, b) = AS$

Переобозначим сотояния: $A \Rightarrow A$, $H \Rightarrow H$, $AS \Rightarrow Y$, $S \Rightarrow X$.

Два заключительных состояния X и Y сведем в одно заключительное состояние S, используя маркер конца цепочки $\perp$. После такой модификации получаем ДКА с единственным заключительным состоянием S и функцией переходов:

$\delta_1(H, a) = X$
$\delta_1(H, b) = A$
$\delta_1(X, b) = Y$
$\delta_1(X, =) \perp S$
$\delta_1(A, a) = Y$
$\delta_1(Y, a) = Y$
$\delta_1(Y, b) = Y$
$\delta_1(Y, \perp) = S$

По ДКА строим грамматику G_1, позволяющую воспользоваться алгоритмом детерминированного разбора:

G_1:

$S \rightarrow X\perp \mid Y\perp$
$Y \rightarrow Ya \mid Yb \mid Aa \mid Xb$
$X \rightarrow a$
$A \rightarrow b$

3.7 Регулярные выражения

Регуля́рные выраже́ния (англ. *regular expressions*) — формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов (символов-джокеров, англ. *wildcard characters*). По сути это строка-образец (англ. *pattern*, по-русски её часто называют «шаблоном», «маской»), состоящая из символов и метасимволов и задающая правило поиска.

Для задания любого регулярного языка над заданным алфавитом есть еще один широко используемый в математических теориях и приложениях формализм, задающий регулярные языки. Это регулярные выражения. Они позволяют описать любой регулярный язык над заданным алфавитом,

используя три вида операций: + (**объединение**), · (**конкатенация**), * (**итерация**).

Определение Пусть L, L_1, L_2 - языки над алфавитом Σ.

Тогда будем называть:

- **язык $L_1 \cup L_2$** *объединением* языков L_1 и L_2;

- **язык** $L_1 \cdot L_2 = \{\varphi \cdot \psi \mid \varphi \in L_1, \psi \in L_2\}$ - *конкатенацией* (*сцеплением*) языков L_1 и L_2 (содержит всевозможные цепочки, полученные сцеплением цепочек из L_1 с цепочками из L_2);

- ***i*-ой степенью языка L** назовем язык $L^i = L^{i-1} \cdot L$ для $i > 0$, $L^0 = \{\varepsilon\}$.

Язык $L^* = \bigcup_{n=0}^{\infty} L^n$ назовем *итерацией* языка L.

Определение. Пусть Σ - алфавит, не содержащий символов *, +, **ε**, ∅, (,). Определим рекурсивно *регулярное выражение* **γ** над алфавитом Σ и регулярный язык $L(\gamma)$, задаваемый этим выражением:

1) $a \in \Sigma \cup \{\varepsilon, \emptyset\}$ есть регулярное выражение; $L(a) = \{a\}$ для $a \in \Sigma \cup \{\varepsilon\}$; $L(\emptyset) = \emptyset$;
2) если **α** и **β** - регулярные выражения, то:
   а) (**α**) + (**β**) - регулярное выражение; $L((\alpha) + (\beta)) = L(\alpha) \cup L(\beta)$;
   б) (**α**) · (**β**) - регулярное выражение; $L((\alpha) \cdot (\beta)) = L(\alpha) \cdot L(\beta)$;
   в) (**β**)* - регулярное выражение; $L((\beta)^*) = (L(\beta))^*$;
3) все регулярные выражения конструируются только с помощью пунктов 1 и 2.

Если считать, что операция «+» имеет самый низкий приоритет, а операция «*» - наивысший, то в регулярных выражениях можно опускать лишние скобки.

Примеры регулярных выражений над алфавитом $\{a, b\}$:

$a + b$, $(a + b)^*$, $(aa\,(ab)^*bb)^*$.

Соответствующие языки:

$$L(a + b) = \{a\} \cup \{b\} = \{a, b\},$$

$$L((a + b)^*) = \{a, b\}^*,$$

$L((aa\,(ab)^*bb)^*) = \{\varepsilon\} \cup \{\, aa\, x_1 bb\, aa\, x_2 bb \ldots x_k bb \mid k \geq 1,\ x_i \in \{(ab)^n \mid n \geq 0\}$ для $i = 1, \ldots, k\,\}$.

Существуют алгоритмы построения регулярного выражения по регулярной грамматике или конечному автомату и обратные алгоритмы, позволяющие преобразовать выражение в эквивалентную грамматику или автомат.

Регулярные выражения используются для описания лексем языков программирования, в задачах редактирования и обработки текстов; подходят для многих задач сравнения изображений и автоматического преобразования. Расширенные их спецификации (эквивалентные по описательной силе, но более удобные для практики) входят в промышленный стандарт открытых операционных систем *POSIX* и в состав базовых средств языка программирования ***Perl***.

3.8 Задачи лексического анализа

Лексический анализ **(ЛА) - это первый этап процесса компиляции.**

Лексический анализ важен для процесса компиляции по нескольким причинам:

а) замена в программе идентификаторов, констант, ограничителей и служебных слов лексемами делает представление программы более удобным для дальнейшей обработки;

б) лексический анализ уменьшает длину программы, устраняя из ее исходного представления несущественные пробельные символы и комментарии;

в) если будет изменена кодировка в исходном представлении программы, то это отразится только на лексическом анализаторе.

Обработка лексем в фазе лексического анализа

На этом этапе литеры, составляющие исходную программу, группируются в **отдельные лексические элементы**, называемые ***лексемами***.

Выбор конструкций, которые будут выделяться как отдельные лексемы, зависит от языка и от точки зрения разработчиков компилятора. Обычно принято выделять следующие типы лексем: идентификаторы, служебные слова, константы и ограничители.

Каждой лексеме сопоставляется пара:

⟨ ***тип лексемы, указатель на информацию о ней*** ⟩.

Лексический анализатор (ЛА):

- должен выделить в исходном тексте цепочку символов, представляющую лексему, и проверить правильность ее записи;
- зафиксировать в специальных таблицах для хранения разных типов лексем факт появления соответствующих лексем в анализируемом тексте;
- преобразовать цепочку символов, представляющих лексему, в пару ⟨ ***тип лексемы, указатель на информацию о ней*** ⟩;
- удалить пробельные литеры и комментарии.

Очевидно, что лексемы можно описать с помощью регулярных грамматик. Поскольку лексемы не пусты, для описания лексического состава любого языка программирования достаточно автоматных грамматик без **ε**-правил.

Например, идентификатор (*I*) описывается так:

$I \rightarrow a \mid b \mid ... \mid z \mid Ia \mid Ib \mid ... \mid Iz \mid I0 \mid I1 \mid ... \mid I9$

Целое без знака (N):

$N \rightarrow 0 \mid 1 \mid ... \mid 9 \mid N0 \mid N1 \mid ... \mid N9$ и т. д.

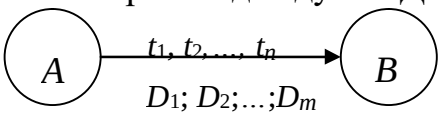
Для грамматик этого класса, как мы уже видели, существует простой и эффективный алгоритм анализа того, принадлежит ли заданная цепочка языку, порождаемому этой грамматикой. Однако перед лексическим анализатором стоит более сложная задача:

- он должен сам выделить в исходном тексте цепочку символов, представляющую лексему, и проверить правильность ее записи;
- зафиксировать в специальных таблицах для хранения разных типов лексем факт появления соответствующих лексем в анализируемом тексте;

преобразовать цепочку символов, представляющих лексему, в пару ⟨ *тип_лексемы, указатель_на_информацию_о_ней* ⟩;

- удалить пробельные литеры и комментарии.

Для описания лексем на основе регулярных грамматик при анализе с помощью диаграммы состояний, введем на дугах дополнительный вид пометок - **пометки-действия**. Теперь каждая дуга в ДС может выглядеть так:

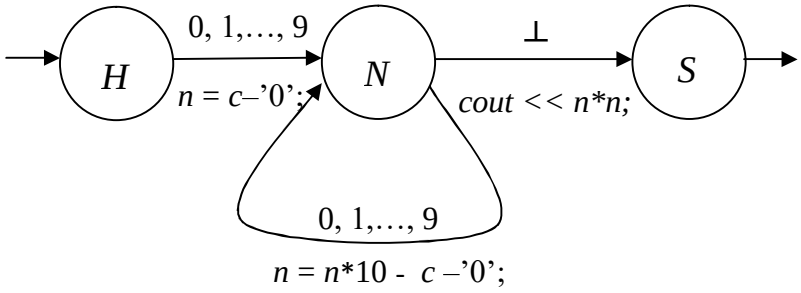


Смысл t_i прежний: если в состоянии A очередной анализируемый символ совпадает с t_i для какого-либо $i = 1, 2, \ldots, n$, то осуществляется переход в состояние B, при этом необходимо выполнить действия $D_1, D_2, ..., D_m$.

Пусть задана регулярная грамматика, описывающей целое число без знака: $S \rightarrow N\perp$

$N \rightarrow 0 \mid 1 \mid ... \mid 9 \mid N0 \mid N1 \mid ... \mid N9$

Необходимо построить диаграмму с действиями по нахождению и печати квадрата данного числа.



Перевод числа во внутренне представление (переменная n типа *int*) по схеме Горнера: распознав очередную цифру, умножаем текущее значениие n на 10 и добавляем числовое значение текущей цифры (текущий символ хранится в переменной c типа *char*). Встретив маркер конца, печатаем квадрат числа n.

3.9 Лексический анализатор для М-языка

Грамматика модельного языка (М- языка)

P → ***program*** D_1; $B\perp$
D_1 → ***var*** D {,D}
D → I {, I}: [***int*** | ***bool***]
B → ***begin*** S {;S} ***end***
S → I := E | ***if*** E ***then*** S ***else*** S | ***while*** E ***do*** S | B | ***read*** (I) | ***write*** (E)
E → E_1 [= | < | > | !=] E_1 | E_1
E_1 → T {[+ | - | ***or***] T}
T → F {[* | / | ***and***] F}
F → I | N | L | ***not*** F | (E)
L → ***true*** | ***false***
I → C | IC | IR
N → R | NR
C → a | b | ... | z | A | B | .. | Z
R → 0 | 1 | 2 | ... |9

Замечания к грамматике модельного языка:

а) запись вида {α} означает итерацию цепочки α, т. е. в порождаемой цепочке в этом месте может находиться либо ε, либо α, либо α α, либо α α α и т. д.;

б) запись вида [α | β] означает, что в порождаемой цепочке в этом месте может находиться либо α, либо β;

в) P - цель грамматики; символ $\perp$ - маркер конца текста программы.

Контекстные условия:

1. Любое имя, используемое в программе, должно быть описано и только один раз.
2. В операторе присваивания типы переменной и выражения должны совпадать.
3. В условном операторе и в операторе цикла в качестве условия возможно только логическое выражение.
4. Операнды операции отношения должны быть целочисленными.
5. Тип выражения и совместимость типов операндов в выражении определяются по обычным правилам; старшинство операций задано синтаксисом.

В любом месте программы, кроме идентификаторов, служебных слов и чисел, может находиться произвольное число пробелов и комментариев в фигурных скобках вида {‹ *любые символы, кроме* «}» и «$\perp$»› }.

true, ***false***, ***read*** и ***write*** — служебные слова (в М-языке их нельзя переопределять в отличие от аналогичных стандартных идентификаторов в Паскале).

Разделители между идентификаторами, числами и служебными словами употребляются так же, как в Паскале.

На вход лексического анализатора поступает текст исходной программы на **М-языке**. Результат - исходная программа в виде последовательности внутреннего представления **лексем**.

Типы лексем

TW — таблица служебных слов М-языка;

TD — таблица ограничителей М-языка;

TID — таблица идентификаторов анализируемой программы.

Таблицы TW и TD заполняются заранее, т. к. их содержимое не зависит от исходной программы; TID формируется в процессе анализа. Содержательно внутреннее представление лексем - это пара (***тип_лексемы, значение_лексемы***).

Вход лексического анализатора - символы исходной программы на М-языке; результат работы - исходная программа в виде последовательности лексем (их внутреннего представления). В нашем случае лексический анализатор будет вызываться, когда потребуется очередная лексема исходной программы, поэтому результатом работы лексического анализатора будет очередная лексема анализируемой программы на М-языке.

Соглашение об используемых таблицах лексем:

TW — таблица служебных слов М-языка;

TD — таблица ограничителей М-языка;

TID — таблица идентификаторов анализируемой программы;

Таблицы TW и TD заполняются заранее, т. к. их содержимое не зависит от исходной программы; TID формируется в процессе анализа.

Необходимые таблицы можно представить в виде объектов, конкретный вид которых мы рассмотрим чуть позже, или в виде массивов строк, например, для служебных слов.

Диаграмма состояний (ДС) для М-языка.

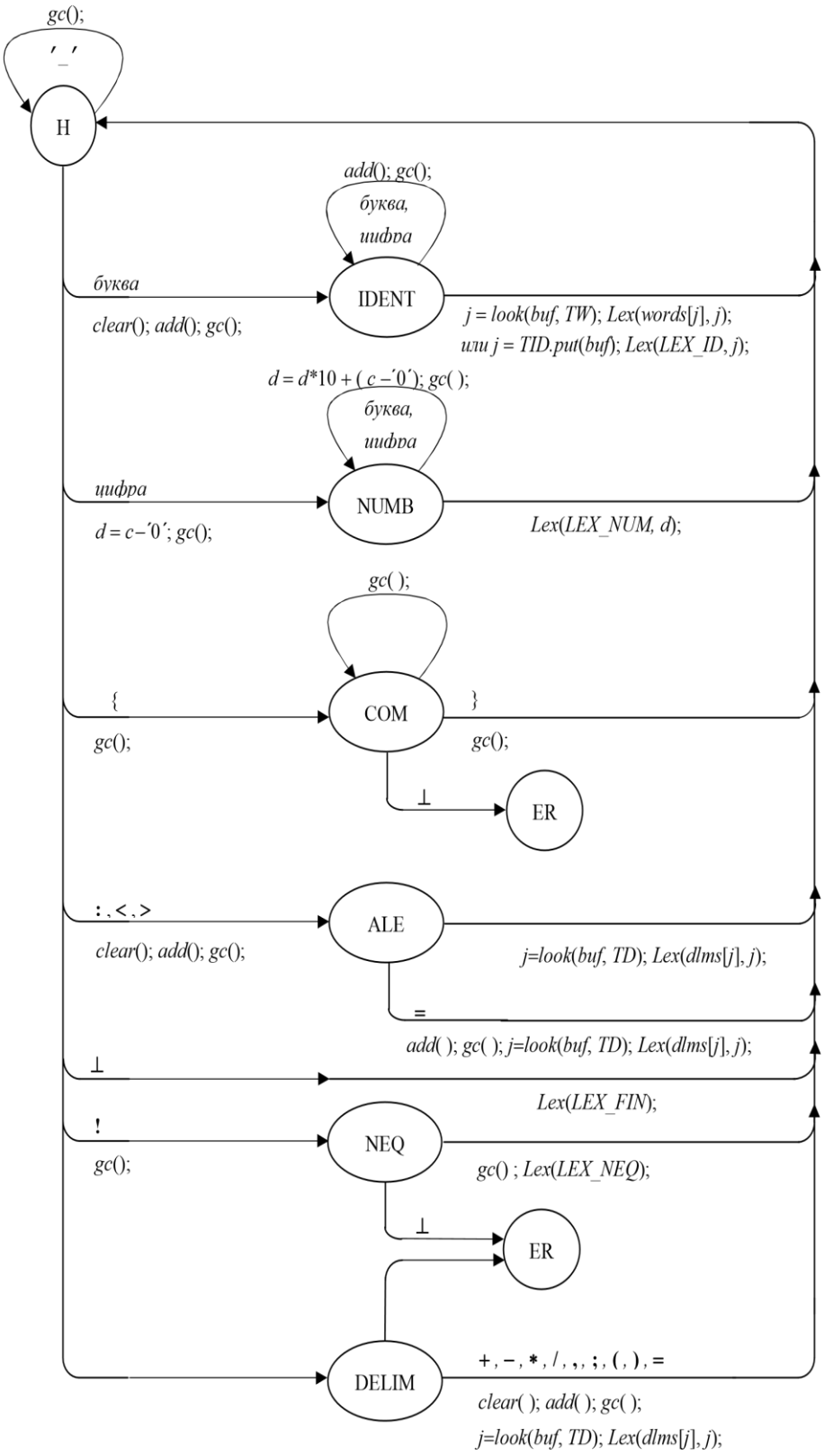
Лексический анализ может проводиться независимо от последующих этапов трансляции. В этом случае исходный файл с текстом программы преобразуется в последовательность лексем во внутреннем представлении согласно построенной ДС; затем эта последовательность подается на вход синтаксическому анализатору.

Мы реализуем другой подход: лексический анализатор выдает очередную лексему по требованию синтаксического анализатора и затем «замирает», пока к нему не обратятся за следующей лексемой.

При таком подходе действие *Lex*(*k, l*)*;* в ДС означает ***return*** *Lex*(*k, l*)*;*. Кроме того, вместо перехода в состояние ошибки ER генерируется исключение с указанием символа, который привел к ошибочной ситуации.

Перехватываться и обрабатываться исключения будут не в лексическом анализаторе, а в основной программе, содержащей анализатор. Обработка исключения состоит в выводе сообщения об ошибке и корректном завершении программы.

Непосредственно реализует ЛА по ДС функция get_lex():



4. Синтаксический анализ

Синтаксический анализ основывается на разных принципах, и использует различные техники построения дерева вывода. Каждый метод предполагает свой способ построения по грамматике программы-анализатора, которая будет осуществлять разбор цепочек. Корректный анализатор завершает свою работу для любой входной цепочки и выдает верный ответ о принадлежности цепочки языку. Анализатор некорректен, если:

- не распознает хотя бы одну цепочку, принадлежащую языку;
- распознает хотя бы одну цепочку, языку не принадлежащую;
- зацикливается на какой-либо цепочке.

Обычно для синтаксического разбора используют КС-грамматики, **правила которых имеют вид $A \to \alpha$, где $A \in N$, $\alpha \in (T \cup N)$**. Грамматики этого класса, с одной стороны, позволяют вполне адекватно описать синтаксис реальных языков программирования; с другой стороны, для разных подклассов КС-грамматик построены достаточно эффективные алгоритмы разбора, расходующие на обработку входной цепочки линейно сопостовимое время.

Говорят, что метод анализа применим к данной грамматике, если анализатор, построенный в соответствии с этим методом, корректен.

Из теории синтаксического анализа известно, что существует алгоритм, который по любой данной КС-грамматике и данной цепочке выясняет, принадлежит ли цепочка языку, порождаемому этой грамматикой. Но время работы такого алгоритма (синтаксического анализа с возвратами) экспоненциально зависит от длины цепочки, что с практической точки зрения совершенно неприемлемо.

Существуют табличные методы анализа, применимые ко всему классу КС-грамматик и требующие для разбора цепочек длины n времени Cn^3 (алгоритм Кока-Янгера-Касами), где C — константа, либо Cn^2 (алгоритм Эрли). Их разумно применять только в том случае, если для интересующего нас языка не существует грамматики, по которой можно построить анализатор с линейной временной зависимостью от длины цепочки (такими языками могут быть, например, подмножества естественного языка).

При разработке языков программирования их синтаксис обычно стараются сделать таким, чтобы время на анализ было прямо пропорционально длине программы. Алгоритмы анализа, расходующие на обработку входной цепочки линейное время, применимы только к некоторым подклассам КС-грамматик.

Сущность синтаксического анализа достаточно сложна, чтобы излагать ее «на пальцах». Даже такое формализованное средство представления как конечный автомат является недостаточно «мощным» для этого. В самом первом приближении это можно показать, ссылаясь на вложенный характер

конструкций языка. Элементы лексики представляют собой линейные последовательности, анализ которых и производится конечными автоматами. Синтаксис же предложения не является линейной структурой. Если определения элементов синтаксиса языка (выражения, операторы) являются теми единицами, из которых строится программа, то взаимоотношение этих единиц в конкретной программе может быть представлено в виде дерева. А с деревом тесно связаны такие понятия, как рекурсия и стек. Таким образом, интуитивно становится ясным, что для определения и анализа синтаксиса языка необходим математический аппарат, который допускает рекурсивное определение и порождение своих элементов, а при их анализе используются деревья, рекурсивные функции и работа со стеком.

Формальные грамматики (ФГ) являются математическим аппаратом, который исследует свойства цепочек символов, порожденных заданным набором правил.

Формальные грамматики (ФГ), и контекстно-свободные и контекстно-зависимые, как математический аппарат появились именно по «производственной необходимости» Представление синтаксиса в компиляторах и **автоматизации синтаксического анализа (СА) на основе схем синтаксически управляемых трансляций** по «производственной необходимости» опирается на математический аппарат формальных грамматик. В отличие от лексики и семантики, которые можно в принципе реализовать, использую только содержательные (неформальные) средства, синтаксический анализ почти невозможно выполнить, руководствуясь «здравым смыслом» и очевидностью. Необходим некоторый промежуточный уровень между синтаксисом языка и программной системой, его распознающей. Таковым уровнем и является ФГ.ФГ является не только средством представления синтаксиса, на ее основе создается инструментарий СА. Самая общая схема синтаксического анализатора, построенного на основе формальных методов.

4.1 Синтаксический разбор. Классификация методов.

Синтаксический разбор (распознавание) является первым этапом синтаксического анализа. Именно при его выполнении осуществляется подтверждение того, что входная цепочка символов является программой, а отдельные подцепочки составляют синтаксически правильные программные объекты. На этапе синтаксического анализа нужно:

1) установить, имеет ли цепочка лексем структуру, заданную синтаксисом языка, и

2) зафиксировать эту структуру.

Следовательно, снова надо решать задачу разбора: дана цепочка лексем, и надо определить, выводима ли она в грамматике, определяющей синтаксис

языка. Если да, то построить вывод этой цепочки или дерево вывода. Однако структура таких конструкций как выражение, описание, оператор и т.п., более сложная, чем структура идентификаторов и чисел.

Вслед за распознаванием отдельных подцепочек можно осуществлять анализ их семантической корректности на основе накопленной информации. Затем проводится добавление новых объектов в объектную модель программы или в промежуточное представление.

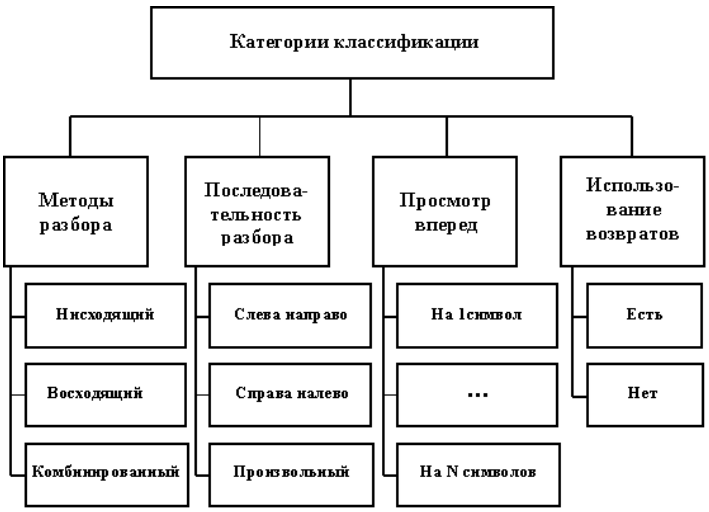
Смысл синтаксического распознавания заключен в доказательстве того, что анализируемая входная цепочка принадлежит множеству цепочек, порождаемых грамматикой данного языка. Если подцепочка не является синтаксически правильным программным объектм, естественно на этом распознавание заканчивается. Выполнение синтаксического распознавания называется **распознаванием входной цепочки**. Цель доказательства в том, чтобы ответить на вопрос: принадлежит ли анализируемая цепочка множеству правильных цепочек заданного языка. Ответ "да" дается, если такая принадлежность установлена. Единственный ответ "нет" на любом уровне ведет к отрицанию принадлежности.

Чтобы получить ответ "да" относительно всей цепочки, надо его получить для каждого правила, обеспечивающего разбор отдельной подцепочки. Так как множество правил образуют иерархическую структуру, возможно с рекурсиями, то процесс получения общего положительного ответа можно интерпретировать как сбор по определенному принципу ответов для листьев, лежащих в основе дерева разбора, что дает положительный ответ для узла, содержащего эти листья. Далее анализируются обработанные узлы, и уже в них полученные ответы складываются в общий ответ нового узла. И так далее до самой вершины.

Классификация методов синтаксического разбора

Дан язык L(G) с грамматикой G = ⟨ T, N, P, S ⟩. Построить дерево разбора входной цепочки

$$a = a_1 a_2 a_3 \ldots a_n.$$



Классификация методов организации синтаксического разбора

Если попытаться формализовать задачу на уровне элементарного метаязыка, то она будет ставиться следующим образом. Дан язык L(G) с грамматикой G = ⟨ T, N, P, S ⟩. Построить дерево разбора входной цепочки **a = $a_1a_2a_3...a_n$.**

Естественно, что существует огромное количество путей решения данной задачи, и целью разработчика распознавателя является выделение приемлемых вариантов его реализации. Общая классификация рассматриваемых вариантов построения распознавателя представлена на слайде.

На самом верхнем уровне выделяются:

- методы разбора;
- последовательность разбора;
- использование просмотра вперед;
- использование возвратов.

Методы разбора

Выделяются два основных метода синтаксического разбора:

- нисходящий разбор;
- восходящий разбор.

Кроме этого можно использовать комбинированный разбор, сочетающий особенности двух предыдущих.

Сам принцип нисходящего и восходящего подхода широко используются в различных областях человеческой деятельности, особенно в тех из них, которые связаны с анализом и синтезом искусственных систем. Например, методы разработки программного обеспечения сверху-вниз (нисходящий) и снизу-вверх (восходящий).

Нисходящий разбор заключается в построении дерева разбора, начиная от корневой вершины. Разбор заключается в заполнении промежутка между начальным нетерминалом и символами входной цепочки правилами, выводимыми из начального нетерминала. Подстановка основывается на том факторе, что корневая вершина является узлом, состоящим из листьев, являющихся цепочкой терминалов и нетерминалов одного из альтернативных правил, порождаемых начальным нетерминалом. Подставляемое правило в общем случае выбирается произвольно. Вместо новых нетерминальных вершин осуществляется подстановка выводимых из них правил. Процесс протекает до тех пор, пока не будут установлены все связи дерева, соединяющие корневую вершину и символы входной цепочки, или пока не будут перебраны все возможные комбинации правил.Построение дерева разбора подтверждает принадлежность входной цепочки данному языку. При этом, в общем случае, для одной и той же входной цепочки может быть построено несколько деревьев разбора. Это говорит о том, что грамматика данного языка является недетерминированной.

Дерево синтаксического анализа (синтаксическое дерево)

Система подстановок (непосредственных выводов) в такой ФГ образует древовидную структуру – **дерево синтаксического анализа (ДСА)**:

- каждому правилу грамматики соответствует поддерево, в котором символ левой части образует вершину-предка, а символы правой части – вершины-потомки;
- ·непосредственному выводу с заменой левой части на правую соответствует процесс построения дерева «сверху-вниз» от предка к потомкам;
- непосредственному выводу с заменой правой части правила на левую соответствует достраивание вершины – предка над группой вершин – потомков;
- корневой вершиной ДСА является вершина с начальным нетерминалом **S**.
- конечными (терминальными) вершинами ДСА являются вершины, содержащие терминальные символы грамматики;
- последовательность терминальных вершин, обойденная слева направо, образует предложение языка, для которого построено ДСА.

Синтаксическое дерево является не просто иллюстрацией последовательности подстановок (выводов). Оно является единственным результатом синтаксического анализа. Фактически оно выявляет все структурные характеристики транслируемого текста, такие как вложенность или приоритеты отдельных синтаксических элементов. Порядок обхода синтаксического дерева (или то же самое, что порядок его построения) определяют последовательность выполнения операций в транслируемой программе, соответствующих отдельным правилам грамматики.

В практике синтаксического анализа используются только контекстно-свободные грамматики. Кроме формальных оснований (сложность и эффективность алгоритмов работы анализаторов) этому способствуют и особенности содержательной работы с такими грамматиками:

система подстановок (непосредственных выводов) в такой ФГ образует древовидную структуру – **дерево синтаксического анализа (ДСА) или синтаксическое дерево**:

Справедливости ради следует заметить, что «ветвление» дерева происходит только в том случае, если в правой части правила имеется несколько нетерминалов. В противном случае (при наличии единственного нетерминала) получается линейная цепочка, как это имеет место в регулярных грамматиках.

Использование синтаксического дерева не означает, что оно обязательно существует в распознавателе в виде структуры данных. Как раз наоборот. Большинство распознавателей в процессе работы производят последовательный выбор правил, соответствующий процедуре обхода дерева, при этом само дерево разворачивается «во времени». Здесь имеется

прямая аналогия с рекурсивными функциями: дерево «экземпляров» представляет собой развернутую во времени последовательность вызовов функцией самой себя, а стек в каждый момент времени содержит локальный контекст (фреймы) текущей последовательности вызовов. Иногда, правда, синтаксическое дерево может быть построено, но уже как результат работы распознавателя.

И, наконец, из рекурсивного характера система правил и процесса их подстановки следует наличие **стека** в распознавателе. Но есть еще одно более жесткое утверждение: кроме стека распознаватель не нуждается более ни в какой дополнительной изменяемой памяти, кроме «зашитых» в нем управляющих таблиц, т.е. является **конечным автоматом,** работающим со входной строкой – анализируемым предложением языка. Естественно, что система команд такого распознавателя предполагает стандартные действия со стеком, продвижение по входной строке и систему состояний-переходов, зависящих от текущих символов стека и входного предложения. Содержимое стека в различных распознавателях может быть разным, но имеющим один и тот же смысл по отношению к синтаксическому дереву: это граница его **недостроенной части.**

4.2 Постановка задачи синтаксического анализа

Понятно, что любая задача преобразования цепочек символов в рамках любой грамматики может быть решена путем полного перебора всех возможных вариантов подстановок, что влечет за собой экспоненциальную (показательную) трудоемкость решения задачи, не приемлемую в реальных условиях. Любой человек, использующий транслятор, в праве ожидать, что последний не слишком уменьшает скорость работы при росте объема транслируемой программы (то есть трудоемкость близка к линейной). Для этого прежде всего требуется отказаться от «тупого» перебора вариантов подстановки с возвратами к промежуточным цепочкам и обеспечить на каждом шаге выбор единственно правильного направления движения из нескольких возможных (так называемый **жадный алгоритм**).

Теперь следует разобраться, в каких взаимоотношениях находятся формальные грамматики и синтаксический анализ.

Синтаксис любого языка программирования определяется формальной грамматикой (контекстно-свободной в нашем случае) - системой терминальных и нетерминальных символов и множеством правил. Анализируемая программа представляется предложением языка. **Задача синтаксического анализа** - определить, является ли это предложение правильным и построить для него последовательность непосредственных выводов из начального символа**,** или синтаксическое дерево.

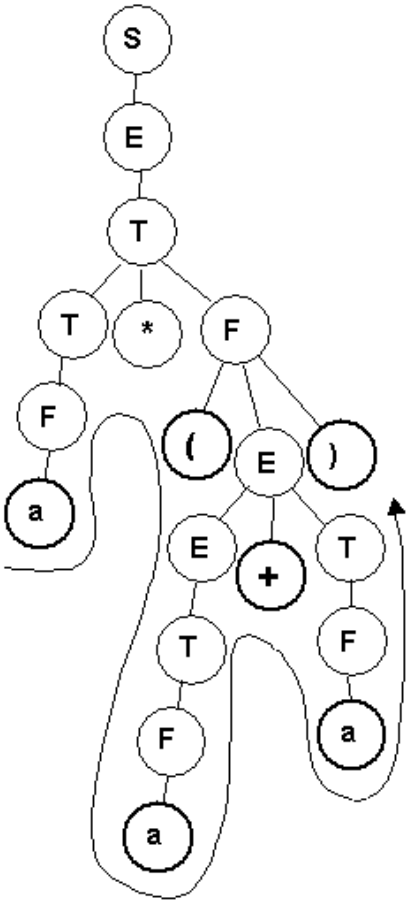
Сам процесс построения дерева, равно и синтаксического анализа может быть как **нисходящим,** т.е. от вершины-предка к вершинам-потомкам с заменой левых частей правил на правые, и наоборот, **восходящим.**

В этом же контексте объясняется понятие **синтаксической ошибки.** Если на каком-то этапе построения синтаксического дерева встречается недопустимая или тупиковая ситуация, то построенная последовательность терминальных символов соответствует синтаксически правильной части программы, а очередной «незакрытый» терминальный символ локализуется как синтаксическая ошибка.

Сам процесс должен быть однозначным, т.е. каждому правильному предложению должно соответствовать единственное ДСА, а процесс его построения не должен содержать «возвратов», т.е. на каждом шаге алгоритма распознавания при наличии альтернативных вариантов должен выбираться каждый раз единственно верный.

Пример дерева синтаксического анализа для распознавания цепочки вида **a*(a+a).**

N = {S, E, T, F},
T = {+,-,/,*,(,),a}
Z → E
E → E + T | E – T | T
R → T * F | T / F | F
F → a | (E)



1. ДСА имеет единственно возможный вид, представленный на рисунке.

2. Обход терминальных вершин слева-направо соответствует распознаваемому предложению.

3. Размещение вершин соответствует приоритету операций, менее приоритетные соответствуют поддеревьям, расположенным ближе к корню.

4. Последовательность выполнения операций можно получить, используя рекурсивный обход ДСА и предполагая, что каждая вершина возвращает результат выполнения действий в своем поддереве (интерпретация) или цепочку команд, производящую эти действия (компиляция).

5. Сам способ построения ДСА (алгоритм синтаксического распознавания) пока обсуждать не будем.

Взаимосвязь синтаксиса и формально определёных грамматик

После предварительного анализа свойств грамматики необходимо напомнить, что формальные грамматики в целом играют роль **языка программирования синтаксиса,** а синтаксис языка программирования, представленный в виде конкретной грамматики – своеобразный аналог исходного текста программы. Здесь необходимо упомянуть тезисы, очевидные для любого программиста:

- одна и та же программа может быть написана различными способами. Точнее, одна и та же идея, воплощенная в результатах работы программы, может быть реализована на формальном языке множеством различных способов;
- программы пишутся «содержательно», не существует формальных методов разработки программ;
- невозможно определить формальными методами, эквивалентны ли две программы с точки зрения результата работы.

Возвращаясь к формальным грамматикам, можно аналогично утверждать:

- один и тот же синтаксис можно реализовать в виде множества грамматик;
- синтаксис преобразуется в ФГ «содержательно»;
- определить, идентичны ли две грамматики с точки зрения реализованного в них синтаксиса, формальными методами невозможно (в терминах теории алгоритмов **задача определения эквивалентности грамматик алгоритмически неразрешима).**

Наиболее продуктивно в синтаксическом анализе используются контекстно-свободные (КС-грамматики).

Независимо от конкретного языка присутствуют такие элементы как **альтернатива** (возможность выбора одной из синтаксических конструкций в данном контексте), **повторение, вложенность, приоритеты, необязательные элементы,** а также имеют место **ограничители, разделители** и т.п. Поэтому прежде всего стоит рассмотреть, как эти элементы синтаксиса реализуются по отдельности. Особенность

использования для этих целей формальных грамматик состоит в том, что в них практически везде используется рекурсия как рабочий инструмент.

Нетерминальный символ – обозначение элемента синтаксиса и места его вхождения (подстановки) в другие элементы синтаксиса.

Кроме нетерминалов, явно обозначающих синтаксические единицы, для многих элементов синтаксиса используются вспомогательные нетерминалы. Иногда они вводятся, исходя из «технологических» соображений для придания грамматикам необходимых необходимых свойств (напр. конфликты «свертка-перенос»)

Альтернатива. Возможность выбора и подстановки одной из множества синтаксических конструкций обеспечивается наличием нескольких правил с одинаковой левой частью, нетерминал которой и обозначает соответствующую синтаксическую конструкцию.

Повторение. Построение повторяющихся цепочек базируется на известном свойстве: линейная (не ветвящаяся) рекурсия эквивалента циклу. В отношении формальных грамматик это означает, что нетерминал, “отвечающий за повторение”, непосредственно или косвенно рекурсивен. Остальные символы правил являются элементами повторения. Завершение процесса повторения должно сопровождаться применением правила, не содержащего такой рекурсии.

Из всех классов формальных грамматик только контекстно-свободные (КС-грамматики) продуктивно используются в синтаксическом анализе. Они дают дополнительную смысловую нагрузку термину "нетерминальный символ"

Общим недостатком курсов теории формальных языков является отрыв грамматик от продуцируемого синтаксиса. Обычно грамматика изучается как некоторая «данность», а не результат программирования определенного синтаксиса. Аналогично традиционным приемам, используемом в обычном программировании, в программировании синтаксиса независимо от конкретного языка присутствуют такие элементы как **альтернатива** (возможность выбора одной из синтаксических конструкций в данном контексте), **повторение, вложенность, приоритеты, необязательные элементы,** а также имеют место **ограничители, разделители** и т.п. Поэтому прежде всего стоит рассмотреть, как эти элементы синтаксиса реализуются по отдельности. Особенность использования для этих целей формальных грамматик состоит в том, что в них практически везде используется рекурсия как рабочий инструмент.

В качестве примера рассмотрим определение идентификатора:

Использование **повторения для** определения идентификатора:

$I \rightarrow LX$

$X \rightarrow \varepsilon$ завершение повторения

$X \rightarrow LX \mid DX$ повторение – прямая рекурсия

$L \rightarrow a \mid \ldots \mid z$

$D \rightarrow 0 \mid \ldots \mid 9$

Поскольку повторение часто используется в синтаксисе, существует еще одна, неканоническая форма представления правил с использованием метасимволов повторения. Круглые и фигурные скобки используются как метасимволы для обозначения ненулевого и любого, соответственно, числа повторений цепочки, заключенной в них. Кроме того, вертикальная черта внутри них может использоваться как обозначение выбора (альтернативы) одной из нескольких цепочек.

E→ T {+T | -T} цепочка сложений/вычитаний
C→ (D) десятичная константа
I → L{L | D} идентификатор

Вложенность. Принцип вложенности одних синтаксических конструкций в другие естественным образом реализуется в грамматике: нетерминал в левой части правила обозначает синтаксическую единицу, которая “расшифровывается” правой частью, нетерминалы в правой части обозначают “вложенные” элементы синтаксиса. Вложенность может быть и рекурсивной, когда в определении синтаксической единицы используются (прямо или косвенно) синтаксические единицы такого же типа. Пример: синтаксис определения вложенных операторов:

O→ for (E;E;E)O | if (E)O else O | E; | {S}
S→ O | OS

Приоритеты. Наличие приоритетов в цепочке символов (операций) представляет собой не что иное, как неявную вложенность: любая цепочка операций более высокого приоритета образует синтаксическую единицу, которая входит в описание цепочки операций текущего приоритета. Поэтому для каждого уровня приоритета необходима своя группа правил, генерирующая цепочку (последовательность) таких операций.

Так, например, синтаксический элемент – арифметическое выражение (нетерминал **E**) представляет собой группу правил для генерации последовательности операций сложения вычитания, «операндами» в которой выступает нетерминалы следующего уровня – **T**. Они, в свою очередь, являются левой частью для группы правил, генерирующих последовательность операций умножения и деления для «операндов» следующего уровня – термов (**F**). То есть каждый элемент суммы может (потенциально) быть произведением и т.д. Естественно, что должно быть предусмотрено прямое приведение нетерминала одного уровня к нетерминалу другого при отсутствии соответствующих операций. «Приоритетные» скобки реализуются в виде правила, в котором нетерминал последовательности операций низшего приоритета, заключенный в скобки, сводится к нетерминалу высшего приоритета:

E → E + T | E - T | T
T → T * F | T / F | F
F → a | (E)

Аналогично для грамматики, использующей аннулирующие правила:

E → TM

M → ε | +TM | -TM

T → FG

G → ε | *FG | /FG

F → a | (E)

Необязательные элементы синтаксиса – это фрагменты синтаксических конструкций, которые не обязательно могут присутствовать в синтаксическом элементе. Способ их описания может быть разным. При использовании аннулирующего правила необязательная часть обозначения отдельным нетерминалом, который раскрывается либо в виде такой конструкции, либо в виде пустой цепочки:

O → if (E)OX

X → else O | ε оператор if с else и без него

Возможно также явное перечисление вариантов синтаксиса в правых частях правил (например, вызов функции с пустым списком параметров, единственными параметром и их последовательностью):

F → a | a()| a(E) | a(SE)

S → E| SE

Разделители. Терминальный символ-разделитель используется для “подсказки”, что вслед за ним следует еще один повторяющийся элемент синтаксиса. Таким образом, он обозначает повторение рекурсии, поэтому в грамматике, ориентированной на нисходящий разбор, за ним обычно следует рекурсивный нетерминал.

Типичный пример, список, разделенный запятыми (с использованием **аннулирующего правила** – правила с пустой правой частью):

L→ aX

X→ ε завершение списка

X→ ,aX продолжение списка

L => aX => a,aX => a,a,aX => a,a,a

В грамматиках, ориентированных на восходящий разбор, разделитель совместно с первым элементом перечисления (списка) используется для введения нетерминала, обозначающего незавершенный список:

L→ a | Sa S – обозначение незавершенного списка

S→ a, | Sa,

a,a,a => Sa,a => Sa => L

Ограничители. Терминальный символ-ограничитель, наоборот, обозначает завершение процесса повторения при генерации символов как одной, так и несколькими группами правил. При этом сам ограничитель встречается в правой части правила вслед за нетерминалом, из которого

генерируются все повторяющиеся цепочки. Таким образом, ограничитель всегда является контекстом (окружением) “более высокого уровня”, чем сама цепочка. Типичные примеры: список операторов, заключенный в фигурные скобки, имеет ограничителем закрывающуюся скобку:

O → {S}
S → ε завершение списка
S → OS повторение
O => {S} => {OS} => {OOS} => {OOOS} => {OOO}

Технология программирования» синтаксиса в формальных грамматиках

Имея под рукой ряд приемов описания отдельных элементов синтаксиса, можно приступать к разработке ФГ для языка в целом. Эта задача сложна хотя бы в объемном отношении: ФГ реального языка программирования включает в себя несколько сот правил. Здесь опять же проступает аналогия с **технологией программирования:** можно пользоваться «методом северо-западного угла», начиная записывать программу с первого выполняемого оператора, а можно пользоваться нисходящим структурным проектированием. При программировании синтаксиса тоже нужно стремиться к структурированному иерархическому описанию: каждый элемент синтаксиса реализуется отдельной, как можно более независимой группой правил, а входящие в нее нетерминалы обозначают вложенные или «не раскрытые» элементы синтаксиса. Причем по правилам нисходящего проектирования начинать надо с самых общих элементов верхнего уровня.

Аналогичные приемы следует применять для анализа ФГ на предмет продуцируемых ими цепочек: каждая группа правил рассматривается отдельно и анализируется вид генерируемых ею **промежуточных** цепочек, содержащих нетерминалы, соответствующие правилам следующего уровня.

Отношения между символами в формальных грамматиках

Любая ФГ продуцирует бесконечное множество предложений языка. Но, поскольку в них присутствуют внутренние закономерности (обусловленные самой грамматикой), то и между различными символами (как терминальными, так и нетерминальными) существуют отношения, которые характеризуют их взаимное положение в синтаксическом дереве.

Множество FIRST и алгоритм его построения

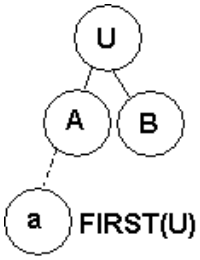
Определим отношения, которые устанавливаются между нетерминальным символом (**промежуточной вершиной ДСА**) и терминальными символами – **конечными вершинами в текущем поддереве и смежных поддеревьях**. Для каждого из нетерминалов входящие в отношение терминальные символы образуют множество. Есть несколько видов множеств, каждому из них можно дать не только формальное определение, но и содержательное название и графическую интерпретацию:

1. множество **FIRST -** «Первых из…»;
2. множество **LAST –** «Последних из…»;
3. множество **FOLLOW** – «Следующих за…» **или последователей.**

Формально множество **FIRST** для заданного нетерминального символа **U** определяется как множество терминальных символов, с которых может начинаться цепочка, выводимая из **U**, т.е.

$$m \in FIRST(U) : U \rightarrow mb$$

Графическая интерпретация – в деревьях с корневой вершиной **U** символы **FIRST(U)** – это крайние правые терминальные символы дерева. Соответственно, если **U** в грамматике обозначает некоторый элемент синтаксиса (выражение, оператор, определение, список параметров и т.п.), то множество **FIRST(U)** на самом деле является ответом на вопрос: «С чего может начинаться (выражение, оператор, определение, список параметров)»? Такое содержательное понимание позволяет в простых случаях построить множество **FIRST**, просто просматривая все возможные последовательности выводов, (учитывая на содержательном уровне продуцируемые правилами повторы и вложенности).



Пример множества **FIRST**

Z → U#
U → U,T | T
T → *T | A
A → Aa | a

1. FIRST(U) : U=>U,T=>U,T,T=>T,… ,T

Символ **U** производит цепочку, состоящую из нетерминалов **T**, разделенных запятыми.

2. T=>*T=>T**
3. T=>Aa=>Aaa=>Aaaa=>aaaa

Символ **T** производит цепочку «звездочек», пока не заменится на символ **A**, который производит цепочку символов **a**. Таким образом, **T** порождает последовательность «звездочек» (возможно, пустую), за которой следует последовательность символов **a**. Отсюда **FIRST(U)={a,*}.**

Если формализовать просмотр возможных правил подстановки, то можно предложить простой рекурсивный алгоритм построения множества **FIRST** для заданного нетерминала **U.**

Простой рекурсивный алгоритм построения множества **FIRST** для заданного нетерминала **U:**

- Просматривается грамматика и выбираются правила с символом **U** в левой части;
- Если в правой части находится терминальный символ, то он добавляется к множеству, т.е. **U→ ab => a ∈ FIRST(U) ;**
- Если в правой части находится нетерминальный символ, то для него также строится множество **FIRST,** которое включается в **FIRST(U),** т.е. **U→ Vb => FIRST(V) ⊆ FIRST(U).**

На практике здесь удобно использовать рекурсивную функцию, которая возвращает строку символов множества**FIRST,** тем более, что принцип рекурсии достаточно ясно отражает суть происходящих событий: для решения задачи с заданным параметром **U** требуется решить аналогичную задачу с другим значением входного параметра. Кроме того, следует учесть рекурсивное «зацикливание» алгоритма для прямой или косвенной левосторонней рекурсии (например, для правила **U→ U,T**). С этой целью рекурсивные вызовы должны передавать ссылку на строку, содержащую все нетерминалы, для которых **FIRST** на данный момент уже строится.

Влияние аннулирующих правил. Множество FIRST*. Описанная выше идиллия нарушается, как только в грамматике появляются правила (нетерминалы), которые могут порождать пустые цепочки.

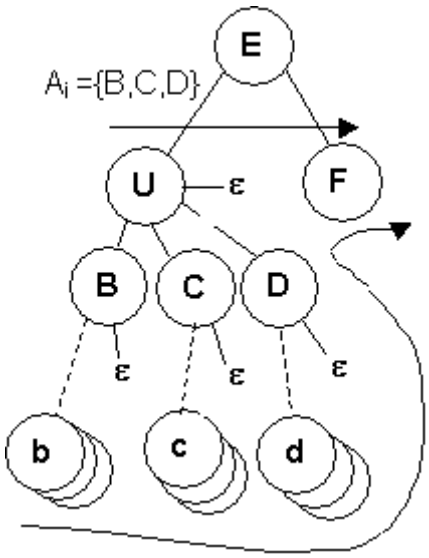
Множество FIRST*.

Когда в грамматике имееются правила (нетерминалы), которые могут порождать пустые цепочки, или **аннулирующие правила (нетерминалы),** тогда «на первое место» выходит следующий за ним символ правила, для которого справедливы все те же действия. Кроме того, аннулирующий нетерминал может порождать пустую цепочку, а может и не порождать, что тоже усложняет алгоритм.

В связи с этим возникает необходимость в **множество FIRST*.** Определим, когда нетерминал является **аннулирующим.** Здесь необходимо учесть, что пустая цепочка может быть произведена не только непосредственно правилом **U→ ε,** но и последовательностью выводов **U => ε.** Свойство «аннулируемости» может быть определено рекурсивно и аналогично вычислено простым рекурсивным алгоритмом - нетерминал **U** порождает пустую цепочку -

- если имеется правило **U→ ε,**

если имеется правило вида **U→ ABC,** в правой части которого находятся только аннулирующие нетерминалы.



Для начала определим, когда нетерминал является **аннулирующим.**

Еще один момент связан с возможностью быть аннулирующим самого нетерминала **U**, для которого строится **FIRST**. Это означает, что сам нетерминал может порождать пустую цепочку, но тогда «на первое место» выходит окружение (контекст), в котором может находиться нетерминал **U**. Поэтому нам придется рассмотреть два случая: множество **FIRST*(U),** построенное без учета контекста, и **FIRST(U),** учитывающий окружение соответствующего нетерминала**.**

Алгоритм построения **FIRST*(U):**

1. Просматривается грамматика и выбираются правила с символом **U** в левой части и непустой правой частью;
2. Выполняется цикл просмотра символов правой части выбранного правила (до первого терминала или до конца);
3. Если очередной символ $\mathbf{A_i}$ является терминальным, то он включается в множество ($\mathbf{A_i} \in$ **FIRST*(U))** и цикл просмотра завершается.
4. Для очередного нетерминала $\mathbf{A_i}$ строится множество **FIRST*(**$\mathbf{A_i}$**),** которое тоже добавляется к **FIRST* (U),** т.е. **FIRST*(**$\mathbf{A_i}$**)** $\subseteq$ **FIRST* (U).**
5. Если $\mathbf{A_i} \Rightarrow \varepsilon$, т.е. нетерминал является аннулирующим, то происходит переход к следующему символу правой части, иначе цикл просмотра завершается.

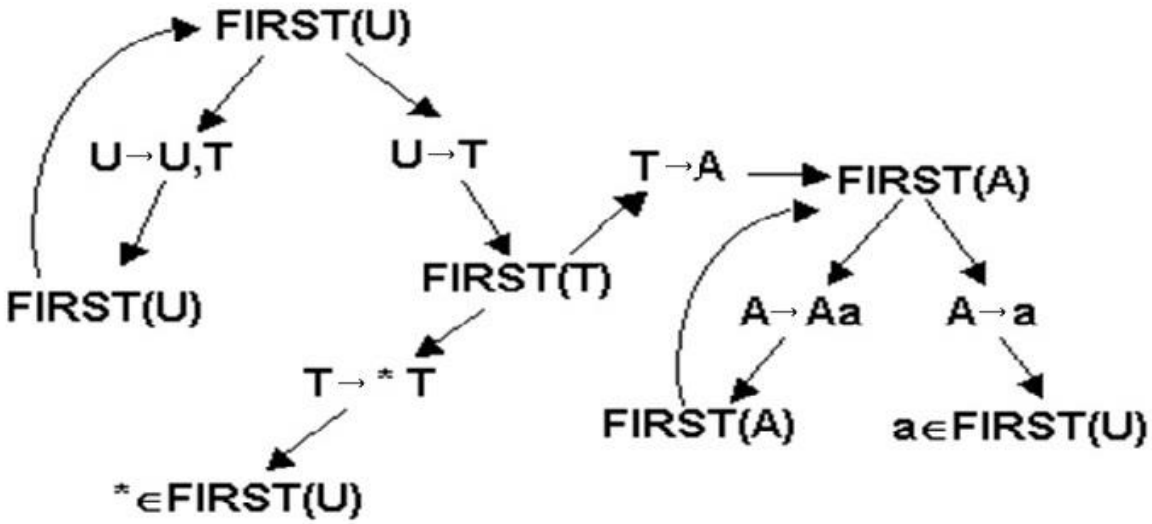
Особенности построения **FIRST(U),** учитывающего окружение (контекст) нетерминала, легко можно увидеть на синтаксическом дереве (см. рисунок выше). Если нетерминальная вершина **U** порождает пустое поддерево, то необходимо подняться вверх по дереву и найти те терминалы, которые могут следовать за **U.** Ниже мы обсудим все подробности построения множества последователей или **FOLLOW(U).** А пока заметим, что алгоритм построения **FIRST(U)** дополняется еще одним пунктом:

6. Если же при выполнении п.3-5 мы дошли до конца правой части правила, т.е. оно состоит только из аннулирующих нетерминалов, то необходимо подняться вверх по дереву к последователю левой части, т.е. **FOLLOW(U)*** $\subseteq$ **FIRST(U).**

Пример для грамматики без аннулирующих правил.

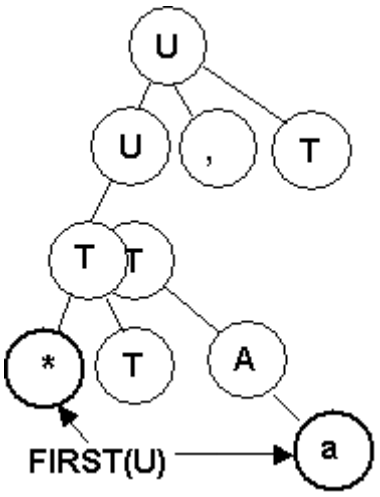
Чтобы «вручную» отследить выполнение рекурсивного алгоритма, можно построить граф (дерево) связей правил и множеств. В его вершины нужно помещать множества **FIRST**, вычисляемые для символов, и правила, на основе которых они вычисляются. Рассмотрим граф для нетерминала уже использованной грамматики:

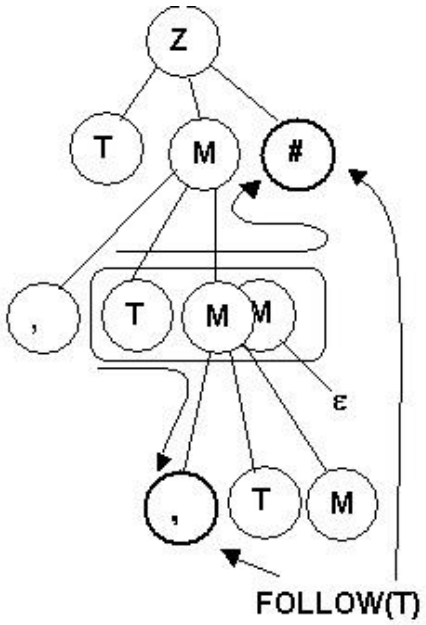
Z→ U#
U→ U, T | T
T→ *T | A
A→ Aa | a



Множество FOLLOW и алгоритм его построения

Легко заметить, что множество **FIRST** связано с построением дерева «в глубину и влево». Образно говоря, оно представляет собой «левый нижний край» дерева, вернее, множества деревьев, выводимых из исходного нетерминала (на рисунке изображена сдвоенная вершина для вариантов построения цепочек **U=>T=>*T** и**U=>T=>A=>a**). С применением аннулирующих правил связано другое «перемещение» - вправо от текущего нетерминала с определением смежных символов. Такое движение «вдоль по правой части правила» с погружением в поддеревья нетерминальных вершин позволяет определить свойство **смежности** двух символов или **следования** одного символа за другим. Формально множество последователей **FOLLOW** для заданного нетерминального символа **U** определяется как множество терминальных символов, которые могут находиться вслед за **U** в произвольной промежуточной цепочке, выводимой из **Z**.





m ∈ FOLLOW(U)
Z ⇒ αUmβ
Z → TM#
M → ,TM | ε
T → *T | A
A → Aa | a

Алгоритм построения **FOLLOW(T)**

1. Просматриваются правые части всех правил грамматики, для каждого найденного символа **T** выполняются следующие действия;
2. Правая часть просматривается от символа, следующего за **T,** до первого терминала (или до конца);
3. Если очередной символ $\mathbf{A_i}$ является терминальным, то он включается в множество ($\mathbf{A_i} \in$ **FOLLOW(T))** и цикл просмотра прекращается;
4. Если очередной символ $\mathbf{A_i}$ является нетерминалом, то для него строится множество $\mathbf{FIRST^*(A_i)}$**,** которое добавляется к **FOLLOW(T),** т.е. $\mathbf{FIRST^*(A_i)} \subseteq$ **FOLLOW(T)**;
5. Если нетерминал $\mathbf{A_i}$ является аннулирующим, т.е. может порождать пустые цепочки, то происходит переход к следующему символу правой части, иначе просмотр завершается;
6. Если при выполнении п.2-5 мы дошли до конца правой части правила, т.е. оно от символа **T** до конца состоит только из аннулирующих нетерминалов (либо этот символ является последним), то необходимо подняться вверх по дереву к последователю левой части, т.е. **FOLLOW(M)** ⊆ **FOLLOW(T),** где **M** –нетерминал левой части.

Графическая интерпретация видна из определения: для нетерминала **T** ищутся ближайшие терминальные вершины «вправо вниз». При поиске учитываются аннулирующие нетерминалы: если вершина может порождать пустую цепочку, то движение вправо продолжается, а по достижении конца текущего уровня (правой части правила) происходит

переход вверх (через символ левой части). Такое образно-интуитивное описание нужно дополнить формальным:

Замечания по алгоритмам построения FIRST и FOLLOW. Несмотря на то, что множества **FIRST** и **FOLLOW**различны по своей сути, их алгоритмы практически идентичны. Единственная разница состоит в следующем:

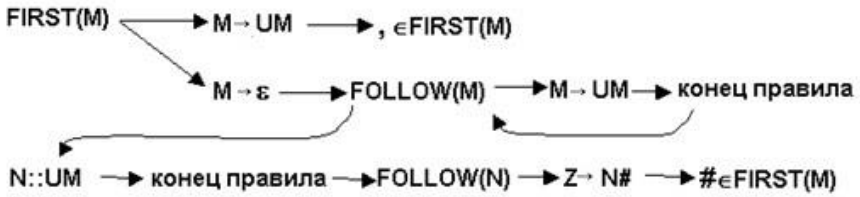
- при построении **FIRST** нетерминал ищется в левых частях правил и просматриваются правые части целиком;

- при построении **FOLLOW** нетерминал ищется в правых частях правил и они просматриваются от найденного символа до конца.

Кроме того, при построении **FOLLOW** всегда учитывается окружающий контекст, т.е. по окончании цикла просмотра правой части производится построение **FOLLOW** для нетерминала левой. Для **FIRST** выделяется как контекстный (**FIRST**), так и неконтекстный (**FIRST***) варианты.

Пример построения множеств FIRST и FOLLOW для простой грамматики. Если Вы заметили, алгоритмы построения похожи, только **FIRST** ищет нетерминал в левых частях, а **FOLLOW** – в правых. Граф-схема связей правил и множеств тоже аналогична.

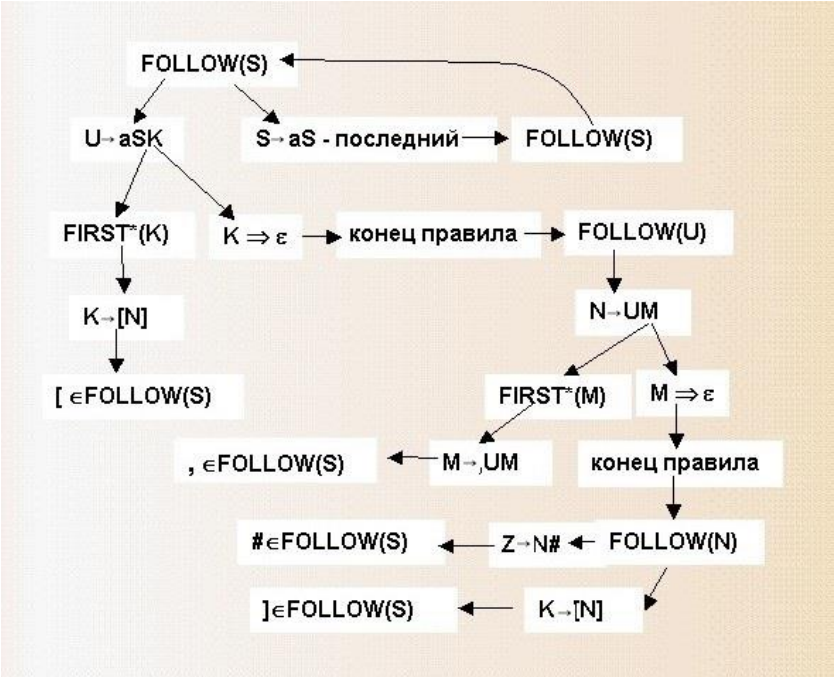
$Z \rightarrow N\#$
$N \rightarrow UM$
$M \rightarrow ,UM \mid M \rightarrow \varepsilon$
$U \rightarrow aSK$
$S \rightarrow aS \mid S \rightarrow \varepsilon$
$K \rightarrow [N] \mid K \rightarrow \varepsilon$



Для начала определим аннулирующие нетерминалы грамматики, таковыми являются **K** и **S,** имеющие аннулирующие правила. Поскольку нет правых частей, состоящих исключительно из аннулирующих нетерминалов, то все другие таким свойством не обладают.

Пример построения множеств FIRST и FOLLOW для грамматики.

Для начала определим аннулирующие нетерминалы грамматики, таковыми являются **K** и **S,** имеющие аннулирующие правила. Поскольку нет правых частей, состоящих исключительно из аннулирующих нетерминалов, то все другие таким свойством не обладают.



Содержательная интерпретация множеств FIRST и FOLLOW

Содержательная интерпретация множеств FIRST и FOLLOW зависит от роли соответствующих нетерминалов в синтаксисе, который реализуется грамматикой. Так, если нетерминал используется как обозначение некоторой синтаксической единицы (или фрагмента синтаксиса), то множество **FIRST** является ответом на вопрос: «С какого символа может начинаться синтаксическая единица»? Иногда на него можно ответить, не проводя приведенных выше построений. Множество **FOLLOW** обычно применимо к аннулирующим правилам, которые могут использоваться для реализации различных элементов синтаксиса:

- если аннулирующее правило используется для ограничения повторения последовательности символов (в паре с другими правилами, обозначающими его продолжение), то множество **FOLLOW** для нетерминала левой части представляет ответ на вопрос: «Какие символы ограничивают повторение элемента синтаксиса, или служат признаком такого окончания»?
- если аннулирующее правило используется для обозначения необязательного элемента синтаксиса, то множество **FOLLOW** для нетерминала левой части представляет ответ на вопрос: «Какие символы могут находиться на месте отсутствующего элемента синтаксиса»?

4.3 Метод рекурсивного спуска

Рассмотрим один из фундаментальных методов разбора, применимый к некоторому подклассу КС-грамматик.

Метод рекурсивного спуска (РС-метод) реализует разбор сверху-вниз и делает это с помощью системы рекурсивных процедур.

Для **каждого нетерминала** грамматики создается своя процедура, носящая **его имя**; ее задача — начиная с указанного места исходной цепочки найти подцепочку, которая выводится из этого нетерминала.

Если подцепочку удалось найти, то работа процедуры считается нормально завершенной и осуществляется возврат в точку вызова.

Если такую подцепочку найти не удается, то процедура завершает свою работу, сигнализируя об ошибке. Это означает, что цепочка не принадлежит языку; разбор

Рассмотрим грамматику $G_1 = \langle\{a, b, c, d\}, \{S, A, B\}, P, S\rangle$, где

P:

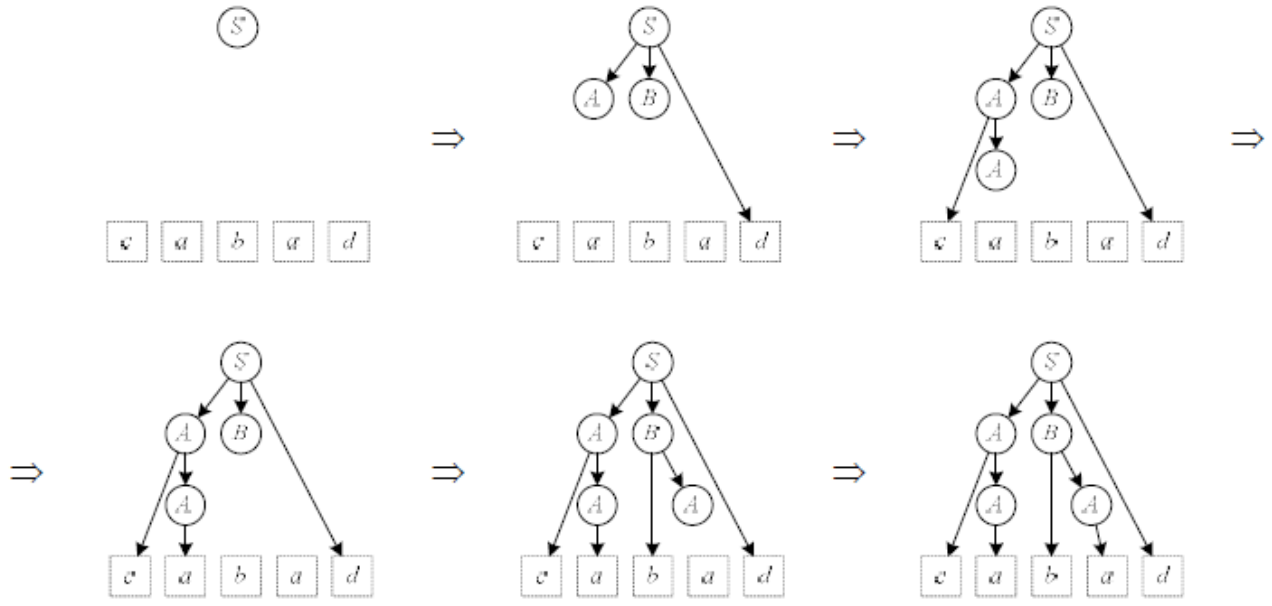
$S \rightarrow ABd$

$A \rightarrow a \mid cA$

$B \rightarrow bA$

и надо определить, принадлежит ли цепочка *cabad* языку $L(G_1)$. Построим левый вывод этой цепочки: $S \rightarrow ABd \rightarrow cABd \rightarrow caBd \rightarrow cabAd \rightarrow cabad.$ Следовательно, цепочка *cabad* принадлежит языку $L(G_1)$.

Построение левого вывода эквивалентно построению дерева вывода методом сверху-вниз (нисходящим методом), при котором на очередном шаге раскрывается самый левый нетерминал в частично построенном дереве.



Метод рекурсивного спуска (РС-метод) реализует разбор сверху-вниз и делает это с помощью системы рекурсивных процедур.

Для **каждого нетерминала** грамматики создается своя процедура, носящая **его имя**; ее задача — начиная с указанного места исходной цепочки найти подцепочку, которая выводится из этого нетерминала.

Если такую подцепочку найти не удается, то процедура завершает свою работу, сигнализируя об ошибке. Это означает, что цепочка не принадлежит языку; разбор останавливается.

Если подцепочку удалось найти, то работа процедуры считается нормально завершенной и осуществляется возврат в точку вызова.

Сформулируем достаточное условие применимости метода рекурсивного спуска.

Достаточное условие применимости метода рекурсивного спуска

Для применимости метода рекурсивного спуска достаточно, чтобы каждое правило в грамматике удовлетворяло одному из двух видов:

(а) $X \to \alpha$, где $\alpha \in (T \cup N)^*$ и это единственное правило вывода для этого нетерминала;

(б) $X \to a_1\alpha_1 \mid a_2\alpha_2 \mid \ldots \mid a_n\alpha_n$,

где $a_i \in T$ для всех $i = 1, 2, \ldots, n$; $a_i \neq a_j$ для $i \neq j$; $\alpha_i \in (T \cup N)^*$, т. е. если для нетерминала X правил вывода несколько, то они должны начинаться с терминалов, причем все эти терминалы должны быть попарно различными;

Это условие не является необходимым. Грамматику, удовлетворяющую данному условию, называют ***s-грамматикой***.

В принципе, по любой грамматике можно построить синтаксический анализатор, но грамматики, используемые на практике, имеют специальную форму.

Метод рекурсивного спуска является одной из возможных реализаций нисходящего анализа с прогнозируемым выбором альтернатив. Прогнозируемый выбор означает, что по грамматике можно заранее предсказать, какую альтернативу нужно будет выбрать на очередном шаге вывода в соответствии с текущим символом (т.е. первым символом из еще не прочитанной части входной цепочки).

Метод рекурсивного спуска (recursive descent method) это один из наиболее простых и, соответственно, один из наиболее популярных методов нисходящего синтаксического анализа.Наша дальнейщая цель - подробно рассмотреть этот подход и сформулировать критерий его применимости.

Принципы, лежащие в основе метода рекурсивного спуска довольно просты. К примеру, рассмотрим задачу вычисления значения арифметической формулы для целочисленных значений, бинарных операций сложения (+), вычитания (–), умножения (*) и деления нацело (/), а также круглых скобок. Как обычно, приоритеты операций умножения и деления равны и их приоритет больше, чем приоритеты операций сложения и вычитания, причем приоритеты этих операций также равны. Будем называть операции + и – операциями типа сложения, а операции * и / – операциями типа умножения. Круглые скобки используются для изменения стандартного порядка выполнения операций. Тогда задача рекурсивного спуска заключается в определении алгоритма, вычисляющго значение формулы.

Отметим, что явно прослеживается связь синтаксического и лексического анализа. Вход синтаксического анализатора –

последовательность лексем и таблицы, например, таблица внешних представлений, которые являются выходом лексического анализатора. Выход синтаксического анализатора – дерево разбора и таблицы, например, таблица идентификаторов и таблица типов, которые являются входом для следующего просмотра компилятора (например, это может быть просмотр, осуществляющий контроль типов). Анализаторы реально используемых языков обычно имеют линейную сложность; это достигается, например, за счет просмотра исходной программы слева направо с заглядыванием вперед на один терминальный символ (лексический класс).

Ну, и последнее замечание. Совсем необязательно, чтобы фазы лексичекого и синтаксического анализа выделялись в отдельные просмотры. Очень часто эти фазы взаимодействуют друг с другом на одном просмотре. Основной фазой такого просмотра считается фаза синтаксического анализа, при этом синтаксический анализатор обращается к лексическому анализатору каждый раз, когда у него появляется потребность в очередном терминальном символе.

4.4 Нисходящий анализ с прогнозируемым выбором альтернатив

В процессе построения левого вывода для произвольной цепочки в грамматике G_1:

$$
\begin{aligned}
S &\rightarrow ABd \\
A &\rightarrow a \mid cA \\
B &\rightarrow bA
\end{aligned}
$$

можно отметить следующее:

(1) любой вывод начинается с применения правила $S \rightarrow ABd$;

(2) если на очередном шаге сентенциальная форма имеет вид $\omega B\alpha$, где $\omega \in T^*$- начало анализируемой цепочки, нетерминал B - самый левый в сентенциальной форме, то для продолжения вывода его нужно заменить на bA (других альтернатив нет);

(3) если на очередном шаге сентенциальная форма имеет вид $\omega A\alpha$, где $\omega \in T^*$- начало анализируемой цепочки, то выбор нужной альтернативы для замены A можно однозначно предсказать по тому, какой символ в анализируемой цепочке следует за начальной подцепочкой ω: если символ α, то применяется альтернатива $A \rightarrow a$, если символ c, то альтернатива $A \rightarrow cA$; если какой-то иной символ - фиксируется ошибка: анализируемая цепочка не принадлежит языку $L(G_1)$.

Если на каком-то шаге получилась сентенциальная форма вида $\omega\alpha$, отличная от (2) и (3), где ω - максимально длинное начало, состоящее только из терминалов, то если α пуста и ω совпадает с анализируемой цепочкой,

процесс вывода успешно завершается, иначе фиксируется ошибка: анализируемая цепочка не принадлежит языку $L(G_1)$.

Еще раз напомним, что сентенциальная форма грамматики ***G*** – это цепочка, выводимая из начального нетерминала грамматики ***G.***

Отмеченные факты по поводу выбора нужной альтернативы на очередном шаге вывода в грамматике G_1 представим в виде так называемой таблицы прогнозов (или таблицы предсказаний):

Таблица прогнозов / предсказаний:

	a	***b***	***c***	***d***
S	$S \to ABd$	$S \to ABd$	$S \to ABd$	$S \to ABd$
A	$A \to a$		$A \to cA$	
B		$B \to bA$		

Имея такую таблицу прогнозов (предсказаний) для КС-грамматики *G*, можно предложить следующий алгоритм нисходящего анализа (построение левого вывода):

1. Начать построение вывода с сентенциальной формы, состоящей из одного начального символа *S*.
2. Пока не будет получена цепочка, совпадающая с анализируемой, повторять следующие действия:

Пусть $\omega Y \alpha$ - очередная сентенциальная форма, где $\omega \in T^*$. Если *w* не совпадает с началом анализируемой цепочки, то прекратить построение вывода и сообщить об ошибке: цепочка не принадлежит $L(G)$. В случае, когда *w* совпадает с началом, и следующим после ω символом в анализируемой цепочке является символ *z*, заменить нетерминал *Y* на правую часть правила, которое находится в ячейке таблицы прогнозов на пересечении строки *Y* и столбца *z*. Если указанная ячейка пуста, прекратить построение вывода и сообщить об ошибке: цепочка не принадлежит $L(G)$.

Как мы видели выше, один из способов реализовать программу-анализатор для нисходящего анализа с прогнозируемым выбором альтернатив заключается в построении системы рекурсивных процедур. Это метод рекурсивного спуска.

Другой способ, с явным использованием стека для хранения нетерминальной части сентенциальной формы, известен как $LL(1)$-анализатор. Техника построения рекурсивных процедур уже была рассмотрена и продемонстрирована на примере, но остался открытым вопрос: как в общем случае «запрограммировать» процедуру на выбор нужной альтернативы по текущему символу. Ответ теперь известен - использовать таблицу прогнозов.

К сожалению, не для каждой КС-грамматики существует таблица с однозначными прогнозами, позволяющая безошибочно осуществить выбор альтернативы на каждом шаге вывода. В некоторых случаях заранее

спрогнозировать выбор альтернативы невозможно: может оказаться, что подходящими в данной ситуации являются сразу несколько альтернатив (неоднозначный прогноз).

Таким образом, нисходящий анализ с прогнозируемым выбором альтернатив пригоден лишь для некоторого подкласса КС-грамматик.

4.5 О применимости метода рекурсивного спуска

Метод рекурсивного спуска **применим** к грамматике, если для нее существует **таблица однозначных прогнозов** и, соответственно, метод рекурсивного спуска (без возвратов) **неприменим к неоднозначным грамматикам**.

Например, по грамматике

G_2:

$S \to aA \mid B \mid d$

A $\to d \mid aA$

B $\to aA \mid a$

нельзя дать однозначный прогноз, что делать на первом шаге при анализе цепочки, начинающейся с символа a, т.е. по текущему символу a невозможно сделать однозначный выбор: $S \to aA$ или $S \to B$.

Множество first **(α)** в грамматике $G = \langle T, N, P, S \rangle$ - это множество терминальных символов, которыми начинаются цепочки, выводимые в G из цепочки $\alpha \in (T \cup N)^*$, т. е. $first(\alpha) = \{a \in T \mid \alpha \Rightarrow a\alpha', \alpha' \in (T \cup N)^*\}$.

Соответственно наличие в грамматике нетерминала X с правилами вида $X \to \alpha$ и $X \to \beta$, из правых частей которых выводятся цепочки, начинающиеся одним и тем же терминалом a, *т.е.* $\alpha \Rightarrow a\alpha'$ и $\beta \Rightarrow a\beta'$, делает неоднозначным прогноз по символу a.

Например, для альтернатив правила $S \to A \mid B$ в грамматике G_3:

$S \to A \mid B$

A $\to aA \mid d$

B $\to aB \mid b$

имеем: $first(A) = \{a, d\}$, $first(B) = \{a, b\}$. Пересечение этих множеств непусто: $first(A) \cap first(B) = \{a\} \neq \emptyset$, и поэтому метод рекурсивного спуска к G_3 неприменим.

Итак, наличие в грамматике правила с альтернативами $X \to \alpha \mid \beta$, такими что $first(\alpha) \cap first(\beta) \neq \emptyset$, делает метод рекурсивного спуска неприменимым.

Рассмотрим пример грамматики, которая может быть **однозначной**, однако **однозначных прогнозов** для нее также ***не существует***:

G_3:

$S \to A \mid B$

C $\to aA \mid d$

D $\to aB \mid b$

Действительно, каждая цепочка, выводимая в G_3 из S, оканчивается либо символом b, либо символом d, и имеет единственное дерево вывода. Но невозможно предсказать, с какой альтернативы ($S \to A$ или $S \to B$) начинать вывод, не просмотрев всю цепочку до конца и не увидев последний символ.

Наличие в грамматике нетерминала X с правилами вида $X \to \alpha A$ и $X \to \alpha B$, из правых частей которых выводятся цепочки, начинающиеся одним и тем же терминалом a, т. е. $\alpha \Rightarrow a\alpha'$ и $\beta \Rightarrow a\beta'$,

делает неоднозначным прогноз по символу a. Соответственно метод рекурсивного спуска **неприменим**.

Как показывает этот пример, грамматика может быть однозначной, однако однозначных прогнозов для нее также не существует. Так что нисходящий анализ с прогнозируемым выбором альтернатив невозможен по такой грамматике, и метод рекурсивного спуска неприменим.

Рассмотрим примеры грамматик с ε-правилами

Примеры **грамматик с ε-правилами**.

G_4:

$S \to aA \mid BDc$	$first(aA) = \{ a \}$, $first(BDc) = \{ b, c \}$;
$A \to BAa \mid aB \mid b$	$first(BAa) = \{ a, b \}$, $first(aB) = \{ a \}$, $first(b) = \{ b \}$;
$B \to \varepsilon$	$first(\varepsilon) = \emptyset$;
$D \to B \mid b$	$first(B) = \emptyset$, $first(b) = \{ b \}$.

Метод рекурсивного спуска неприменим к грамматике G_4, так как

$$first\,(BAa) \cap first\,(aB) = \{a\} \neq \varnothing.$$

Следующий пример показывает еще одно свойство грамматик, наличие которого делает РС-метод неприменимым.

G_5:

$S \to aA$

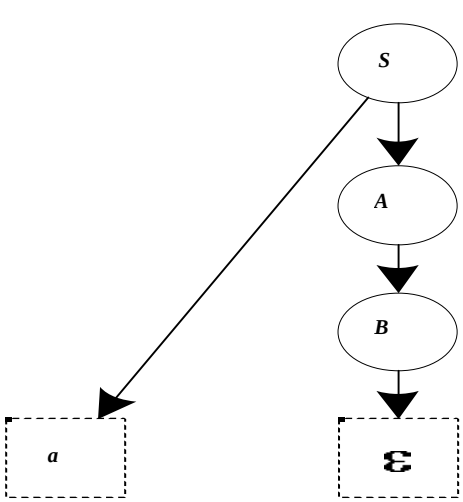
A $\to BC \mid B$

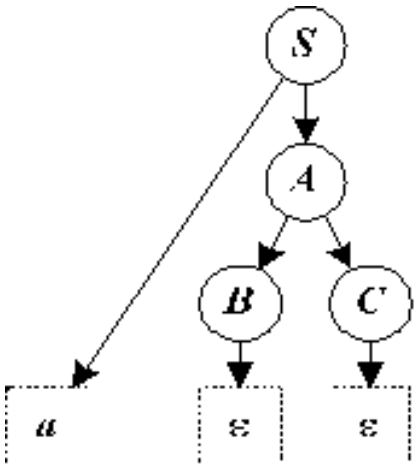
$C \to b \mid \varepsilon$

B $\to \varepsilon$

Пересечение множеств *first* пусто для любой пары альтернатив грамматики G_5, однако наличие двух различных альтернатив, из которых выводится пустая цепочка, делает

данную грамматику неоднозначной и, следовательно, метод рекурсивного спуска к ней неприменим. Действительно, $BC \Rightarrow \varepsilon$ и $B \Rightarrow \varepsilon$. Цепочка a имеет два различных дерева вывода:





Таким образом, если в грамматике для **правила $X \to \alpha \mid \beta$** выполняются **соотношения $\alpha \Rightarrow \varepsilon$ и $\beta \Rightarrow \varepsilon$,** то метод рекурсивного спуска **неприменим**.

Осталось выяснить, как обстоят дела с применимостью метода, если для каждого нетерминала грамматики существует не более одной альтернативы, из которой выводится ε.

Пример грамматики, где для каждого нетерминала грамматики существует не более одной альтернативы, из которой выводится ε.

G_6:

$S \to cAd \mid d$

$A \to aA \mid \varepsilon$

$first(cAd) = \{c\}$, $first(d) = \{d\}$;

Однозначные прогнозы для выбора альтернативы нетерминала S существуют, так как $first(cAd) \cap first(d) = \varnothing$.

Таблица прогнозов для G_6:

	a	***c***	***d***
S		$S \to cAd$	$S \to d$
A	$A \to aA$	$A \to \varepsilon$	$A \to \varepsilon$

Выбор альтернативы для A в данной грамматике также можно однозначно спрогнозировать: если текущим символом является a, применяется правило $A \to aA$, иначе правило $A \to \varepsilon$. Это возможно благодаря тому, что за любой подцепочкой, выводимой из A, следует символ d, который сам в эту подцепочку не входит. Процедура $A(\ )$ при выборе альтернативы $A \to \varepsilon$ просто возвращает управление в точку вызова, не считывая следующий символ входной цепочки. Процедура $S(\ )$, получив управление после вызова $A(\ )$, проверяет, что текущим символом является d. Если это не так, фиксируется ошибка. Конечно, проверку символа d (без считывания следующего символа из входной цепочки) могла бы сделать и сама $A(\ )$, но это излишне, так как $S(\ )$ все равно будет проверять d, и если вместо d обнаружит другой символ, ошибка будет зафиксирована.

Итак, для грамматики G_6, имеющей для каждого нетерминала не более одной альтернативы, из которой выводится пустая цепочка, метод рекурсивного спуска применим.

Процедура $A(\ )$ для нетерминала A, имеющего пустую альтернативу в грамматике G_6, реализуется так:

```
void  A
() {
if ( c =='a' )
{
            cout  <<  "A-
>aA, ";           gc ();
            A ();
      }
      else
      {
            cout << "A->epsilon, "; // след. символ не считывается
      }
}
```

Следующий пример показывает, что наличие альтернативы α, такой что $\alpha \Rightarrow \varepsilon$, все же может сделать метод рекурсивного спуска неприменимым.

G_7:

$$S \to Bd$$
$$B \to cAa \mid a$$
$$A \to aA \mid \varepsilon$$

$first(cAa) = \{c\}$, $first(a) = \{a\}$;

У нетерминала S правая часть единственна и проблема выбора альтернативы для S не стоит. Для выбора альтернативы нетерминала B существуют однозначные прогнозы, поскольку $first(cAa) \cap first(a) = \varnothing$.

Однако для нетерминала A прогноз по символу a неоднозначен. Дело в том, что любой вывод, содержащий A, имеет вид: $S \to Bd \to cAad \to \ldots \to ca\ldots aAad$. Поэтому альтернативу $A \to \varepsilon$ следует применять только тогда, когда текущим символом является a, а следующий за ним символ отличен от a (например, d). Если текущий - a и следующий за ним символ - тоже a, то выбирается альтернатива $A \to aA$. Но сделать однозначный выбор только по

текущему символу в пользу какой-то одной из этих альтернатив невозможно, так как анализатор не умеет заглядывать вперед (в непрочитанную часть анализируемой цепочки).

Как видим, в G_7 существует сентенциальная форма, например *cAad*, в которой после нетерминала *A*, имеющего в грамматике пустую альтернативу, стоит символ *a*, с которого также начинается и непустая альтернатива для *A*. В таком случае процедура *A*( ) не сможет правильно определить по текущему символу *a*, считывать ли следующий символ и вызывать *A*( ) (т. е. применять правило $A \to aA$) или возвращать управление без считывания символа (правило $A \to \varepsilon$). Опишем эту ситуацию более формально.

Итак, на примерах мы рассмотрели все случаи, когда можно построить однозначные прогнозы по грамматике.

Множество *follow(A)* - это множество терминальных символов, которые могут появляться в сентенциальных формах грамматики $G = \langle T, N, P, S \rangle$ непосредственно справа от *A* (или от цепочек, выводимых из *A*), т.е.

$$follow(A) = \{a \in T \mid S \Rightarrow \alpha A \beta,\ \beta \Rightarrow a\gamma,\ \ A \in N,\ \ \alpha, \beta, \gamma \in (T \cup N)\}.$$

Тогда, **если в грамматике есть правило**

$X \to \alpha \mid \beta$**, такое что** $\beta \Rightarrow \varepsilon$,

$first(\alpha) \cap follow(X) \neq \varnothing$**,** то метод рекурсивного спуска **неприменим** к данной грамматике.

Критерий применимости метода рекурсивного спуска.

Подытожив рассмотренные примеры, сформулируем **критерий применимости** метода рекурсивного спуска.

Пусть *G* - КС-грамматика. Метод рекурсивного спуска примени́м к *G*, если и только если для любой пары альтернатив $X \to \alpha \mid \beta$ выполняются следующие условия:

(1) $first(\alpha) \cap first(\beta) = \varnothing$;

(2) справедливо не более чем одно из двух соотношений: $\alpha \Rightarrow \varepsilon$, $\beta \Rightarrow \varepsilon$;

(3) если $\beta \Rightarrow \varepsilon$, то $first(\alpha) \cap follow(X) = \varnothing$.

LL(*k*)- и *LR*(*k*)-грамматики

К классу грамматик, для которых существуют эффективные анализаторы, относятся ***LL*(*k*)-грамматики**, по которым, как правило, реализуется анализ сверху-вниз — нисходящий; ***LR*(*k*)-грамматики**, грамматики предшествования, по которым, как правило, реализуется анализ снизу-вверх — восходящий; и некоторые другие

Анализатор для *LL*(*k*)-грамматик просматривает входную цепочку слева направо и осуществляет детерминированный левый вывод, принимая во внимание *k* входных символов, расположенных справа от текущей

позиции. Выбор альтернативы осуществляется на основе заранее составленной таблицы прогнозов.

Анализатор для *LR*(*k*)-грамматик просматривает входную цепочку слева направо и осуществляет детерминированный правый вывод, принимая во внимание *k* входных символов, расположенных справа от текущей позиции. Вывод строится методом сверток, как при разборе по лево-линейной автоматной грамматике. Предварительно по *LR*(*k*)-грамматике строится таблица, которая на каждом шаге вывода позволяет анализатору однозначно выбрать нужную свертку.

Грамматика *G* является LL(1)-грамматикой тогда и только тогда, когда для любых двух различных ее выводов $A \rightarrow \alpha|\beta$ выполняются следующие условия:

a) Не существует такого терминала *a*, для которого и α, и *β* порождают строку, начинающуюся с *a.*

b) Пустую строку может порождать только один из выводов α или *β*.

c) Если $\beta => \lambda$, то *a* не порождает ни одну строку, начинающуюся с терминала из FOLLOW(*A*).Перечисленные идеи нисходящего разбора соответствуют классу грамматик, именуемому **LL(k),** т.е. идеи эти «работают» на грамматиках соответствующего класса:

- Первая **L** обозначает принцип просмотра входного предложения (цепочки) в направлений слева-направо (**left**) с последовательным «закрытием» символов терминальными вершинами дерева;
- **k** – обозначает глубину просмотра «незакрытой» части цепочки для принятия решения о выборе одной из правых частей правила, соответствующего очередному нетерминалу. Обычно k=1;
- Вторая **L** соответствуют термину «левосторонний вывод» (**left**), т.е. нисходящий разбор с заменой левой части правила на правую.

Какими свойствами должны обладать **LL(1)-грамматики**, будет ясно позднее, после того, как будет изложены принципы организации и алгоритм работы нисходящего распознавателя. Однако основное принцип должен быть соблюден всегда: для каждого нетерминала и очередного незакрытого символа входной строки должна быть обеспечена возможность однозначного и окончательного (безвозвратного) выбора правой части правила, содержащего этот нетерминал в левой части.

5. Семантика формальных языков

Классическая теория формальных языков, как уже отмечалось, занималась исключительно синтаксисом языков, изучая методы порождения и распознавания множеств слов. Семантика формальных языков, сравнительно молодая ветвь теории, занимается способами сопоставления некоего "смысла" словам (цепочкам) языка.

Необходимость в построении точного математического понятия "смысла" диктуется развитием информационных технологий, прежде всего технологии проектирования компиляторов.

Рассмотренные выше языковые модели определенным образом связаны с этапами этой технологии.

Текст входной программы, как известно, анализируется в несколько проходов. На первом проходе производится лексический анализ, а именно проверяется правильность простейших элементов текста, называемых лексемами. Примерами лексем могут служить идентификаторы и константы, разрешенные синтаксисом входного языка программирования. В процессе лексического анализа не проверяется синтаксическая правильность всей программы в целом, а проверяется только синтаксическая правильность лексем (в частности, правильность написания идентификаторов и констант). Так как лексемы обычно являются элементами некоторого регулярного языка, то базовой моделью для лексического анализатора является модель конечного автомата.

Если текст программы успешно прошел этап лексического анализа, то тогда проверяется его глобальная синтаксическая правильность. При этом каждая лексема рассматривается как буква. Здесь применяются методы синтаксического анализа, в частности рассмотренные ранее. В предположении, что синтаксис языка программирования описан КС-грамматикой, основой для построения синтаксических анализаторов, как мы уже видели, выбирается модель синтаксического анализатора.

По завершении синтаксического анализа строится дерево вывода входной программы. После этого переходят к этапу генерации объектного кода, т.е. внутреннего машинного представления входного текста. Это значит, что выполняется перевод с некоторого языка программирования на язык машинных кодов. Но чтобы выполнить перевод текста на другой язык, необходимо каким-то образом понять его "смысл". Следовательно, анализ уже синтаксически проверенной программы с точки зрения ее "смысла" (семантический анализ) необходимо предшествует самой генерации объектного кода. И прежде всего необходимо уточнить математически, что такое "смысл" (как раньше мы математически определяли синтаксис в терминах грамматик).

Т.о. наша задача - рассмотреть возможные методы формального (математического) определения семантики для КС-языков. Тем самым мы

всюду в дальнейшем предполагаем, что язык, семантика которого определяется, может быть задан некоторой КС-грамматикой.

Сразу же необходимо сделать уточнение. Как мы уже заметили ранее, исследуя явление неоднозначности в КС-языках, "смысл" следует сопоставлять не самим словам языка, а деревьям их вывода: меняя дерево вывода данной цепочки, мы меняем и ее "смысл", понимаем ее по-другому. Далее, можно сопоставлять "смысл" не только словам языка, точнее, деревьям вывода этих слов из начального символа грамматики, но и так называемым фразам языка — терминальным цепочкам, выводимым из разных нетерминалов грамматики.

Например, фраза "...а так как мне бумаги не хватило" не является законченным предложением русского языка, но имеет, очевидно, какой-то "смысл". Точно так же оператор присваивания, "вынутый" из какой-то программы на каком-то языке программирования, не является "программой", не может рассматриваться как элемент данного языка, но мы в состоянии сопоставить ему тот или иной "смысл". Тем самым возникает **идея** определить "*смысл*" через **отображение множества "синтаксических объектов"** — деревьев выводов фраз языка в некоторое "предметное множество", **множество "семантических объектов".**

Нашей целью будет элементарное введение в формальное описание семантики КС-языков и описание некоторых простых алгоритмов формального разбора семантики языков программирования.

5.1 Формальное определение фразы КС-языка

Фразой *языка* L(G), *где* G = ⟨T, N, P, S⟩, назовем ***терминальную цепочку*** x, *если* A∈ x *для некоторого нетерминала* A ∈ N.

Если T(x) - дерево вывода фразы x с корневым нетерминалом A, то ***подфразой*** фразы **x** назовём выводимую из некоторого вхождения B терминальную цепочку

Пусть G = ⟨T, N, P, S⟩, - *КС-грамматика.* ***Терминальную цепочку*** **x** *называют фразой языка* L(G), *если* A∈ x *для некоторого нетерминала* A ∈ N.

Допустим T(x) - дерево вывода фразы x с корневым нетерминалом A. Возьмем в T(x) некоторое поддерево с корневым нетерминалом B. Выводимую из этого вхождения Bтерминальную цепочку называют подфразой фразы x. Если вершина дерева T(x), соответствующая данному вхождению B, имеет глубину 1 (или, что равносильно, уровень, на единицу

меньший уровня корня A), то данная подфраза называется подфразой первого уровня фразы x.

Рассмотрим в качестве примера следующую **грамматику G (1)** арифметических выражений:

Expr → Atom | (Expr+Expr) | (Expr∗Expr),

Atom → a_1 | a_2 |…| a_n.

Предполагается, что вместо вхождения нетерминала Atom может быть подставлен любой символ некоторого алфавита V = {a_1, a_2,,…, a_n} (атом в арифметическом выражении может быть либо переменным, либо константой).

Нарисуем дерево вывода выражения (a+(b∗(c+(d∗(e+g))))), где a,b,c,d,e,g - некоторые атомы из V.

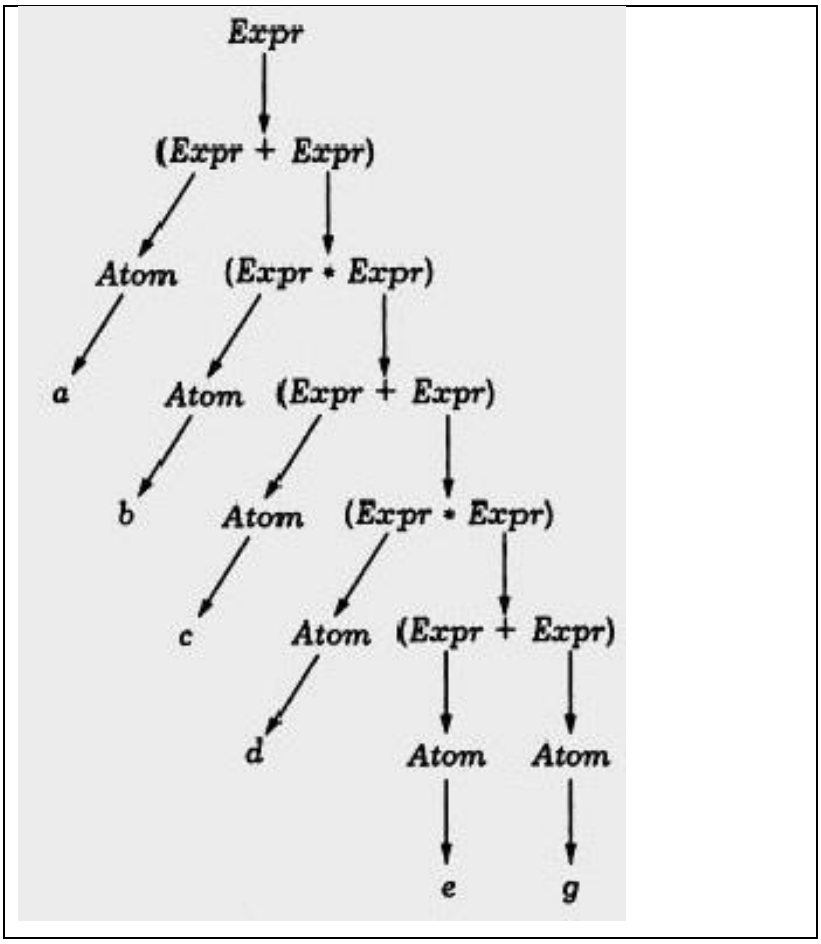


рис. 5.1 Дерево вывода выражения (a+(b∗(c+(d∗(e+g)))))

Это выражение в качестве подфраз первого уровня имеет цепочки a и (b∗(c+(d∗(e+g)))). Вторая фраза имеет подфразы первого уровня b и (c+(d∗(e+g))) и т.д. Заметим, что фраза a, выводимая из самого левого узла Expr глубины 1 имеет в качестве подфразы первого уровня ту же цепочку a, т.е. подфраза первого уровня может совпадать с самой исходной фразой.

Множество L(G) **всех фраз языка** определим как **PH(L)**, а множество **всех деревьев вывода фраз L** - определим как **TPH(L)**. Для однозначного языка L(G) эти множества эквивалентны.

5.2 Семантическая функция языка L

Семантическая функция языка **L** *есть отображение* SEM(L):

$$TPH(L) \rightarrow U(L)$$

- множества **всех деревьев вывода фраз L** в некоторое множество U(L), называемое **универсумом** (предметной областью, семантической областью, областью интерпретации) **языка L**.

Замечание. Ради общности следовало бы определить семантическую функцию как частичное отображение, но мы не будем этого делать, вводя для "бессмысленных" фраз специальный "неопределенный" элемент в универсум ("нуль" предметной области, "неопределенный смысл"). Заметим также, что для однозначного языка семантическая функция может быть определена как отображение из множества фраз языка.

Способы определения семантической функции могут быть различными. Нас будет интересовать важный частный случай, когда значение семантической функции на некоторой фразе (под фразой здесь и далее понимается дерево ее вывода, причем разные деревья для одной и той же цепочки рассматриваются как разные фразы) определяется однозначно через значение этой функции на подфразах первого уровня. Более формально: представим фразу x синтаксически как ‘’соединение" своих подфраз первого уровня:

$$x = \varphi(y_1, y_2, \ldots, y_m) = u_1 y_1 u_2 y_2 \ldots u_m y_m u_{m+1},$$

где $\mathbf{u_1,u_2,\ldots,u_{m+1}}$ - некоторые терминальные цепочки. Например, для определенного выше языка арифметических выражений фраза раскладывается следующим образом:

$$(a + (b * (c + (d * (e + g))))) = u_1 a u_2 (b * (c + (d * (e + g))))u_3,$$

где u_1 есть цепочка (, u_2 - цепочка +, а u_3 - цепочка).

Таким образом, любая фраза представляется, вообще говоря, не как простое соединение своих подфраз первого уровня, а как соединение с "прослойками" в виде определенных терминальных цепочек. **Операцию φ**, задающую такие прослойки, **называют часто операцией конкатенации**. Эта операция определяется грамматикой языка.

Определим отображение $\boldsymbol{\psi: U(L)^m \rightarrow U(L)}$ (т.е. некоторая операция на предметной области), такое, что

$$SEM(L)(x)=\psi(SEM(L)(y_1),\ldots,SEM(L)(y_m)).$$

Это ограничение называют также принципом гомоморфной интерпретации. Примем соглашение об обозначении **семантической функции** языка L(G), $[\![\cdot]\!]_L$, опуская индекс, указывающий на язык, если это не ведет к недоразумению. Тогда формула примет вид:

$$[\![x]\!] = [\![\varphi(y_1,\ldots,y_m)]\!] = \psi([\![y_1]\!], \ldots, [\![y_m]\!]).$$

Тогда предполагается, что, если фраза x представлена в виде (7.1), то определено отображение ψ: $U(L)^m \rightarrow U(L)$ (т.е. некоторая операция на предметной области), такое, что

$$SEM(L)(x)=\psi(SEM(L)(y_1),\ldots,SEM(L)(y_m))$$

Это ограничение, накладываемое на семантическую функцию, назовем принципом гомоморфной интерпретации.

Примем теперь некоторые соглашения об обозначениях.

Семантическую функцию языка L будем обозначать $[\![\cdot]\!]_L$, опуская индекс, указывающий на язык, если это не ведет к недоразумению. Тогда равенство можно переписать в виде

$$[\![x]\!] = [\![\varphi(y_1,\ldots,y_m)]\!] = \psi([\![y_1]\!], \ldots, [\![y_m]\!]).$$

Подобные определения семантики фразы через семантику ее подфраз первого уровня называют семантическими правилами языка. Семантические правила соответствуют синтаксическим правилам — правилам исходной КС-грамматики. Так, для приведенной выше грамматики арифметических выражений, полагая U=R, можно записать следующие семантические правила:

$$[\![Expr]\!] = [\![Expr]\!] + [\![Expr]\!].$$

(для синтаксического правила Expr → (Expr+Expr)), т.е. здесь конкатенарной операции φ, такой, что φ(u,v)=(u+v) для любых двух фраз u и v, сопоставляется обычное арифметическое сложение;

⟦Expr⟧ = ⟦Expr⟧ * ⟦Expr⟧.

(для синтаксического правила Expr → (Expr ∗ Expr)) операция - арифметическое умножение;

⟦Expr⟧ = ⟦Atom⟧

(для синтаксического правила Expr → Atom);

⟦Atom⟧= ⟦a_i⟧

(для синтаксического правила Atom→a_i, i=1, …, n).

Строго говоря, в первых двух правилах мы должны различать три разных вхождения одного и того же нетерминала Expr, так как они соответствуют разным деревьям.

Семантическая функция языка арифметических выражений будет определена, если мы зададим значение этой функции на атомах: это естественно ассоциируется с хорошо знакомой процедурой вычисления значения арифметического выражения при подстановке вместо входящих в него переменных конкретных числовых значений (при этом не исключается, что атом может быть константой - обозначением конкретного числа; в таком случае само синтаксическое правило задает сразу значение семантической функции на данном атоме).

Для языка арифметических выражений семантическая функция будет определена, если мы зададим значение этой функции на **атомах**: это естественно ассоциируется с хорошо знакомой процедурой вычисления значения арифметического выражения при подстановке вместо входящих в него переменных конкретных числовых значений.

Для выражения **(a+(b∗(c+(d∗(e+g)))))**, полагая

⟦a⟧=1, ⟦b⟧=2, ⟦c⟧=3, ⟦d⟧=4, ⟦e⟧=5, ⟦g⟧=6,

получим значение всего выражения, равное 95.

Таким образом, в этом конкретном случае семантика фразы есть числовое значение представленного данной фразой арифметического выражения. Мы рассмотрели в качестве универсума множество вещественных чисел и получили одну семантику. Задав универсум как-нибудь иначе (например, как множество комплексных чисел или множество функций некоторого класса), получим совсем другую семантику. Для одного и того же синтаксиса, следовательно, могут быть определены различные семантики (семантические функции).

Разобранный выше на примере языка арифметических выражений подход к определению семантики называют иногда экстенсиональным

подходом. Суть его состоит в том, что явно определяется универсум как множество "внеязыковых" объектов (экстенсионалов) и каждой языковой фразе сопоставляется некоторый экстенсионал. Для разобранного выше примера экстенсионал — это вещественное число. Слово "внеязыковых" взято в кавычки потому, что универсум сам может быть некоторым языком (выступающим тогда по отношению к исходному языку как метаязык). Не исключено даже, что метаязык совпадает с самим определяемым языком - примером могут служить всевозможные толковые словари.

Существует и другой подход к определению семантики, называемый интенсиональным, одной из разновидностей которого является аксиоматический метод определения семантики языка.

Аксиоматический метод предполагает рассмотрение исходного, "объектного" языка, семантика которого определяется, как формальной теории (или формальной системы). Не давая строгого определения формальной теории в его общности, поясним его суть и рассмотрим пример.

Формальная теория задается как некоторый язык, цепочки которого называются в этом случае утверждениями (или предложениями). В этом языке определяется подъязык так называемых доказуемых утверждений: задается некоторое начальное множество утверждений, которые считаются априори доказанными (множество аксиом теории), и задается некоторое множество правил вывода, применяя которые к некоторым утверждениям (в частности, уже доказанным), можно получать новые утверждения. Если мы применяем правило вывода к доказанному утверждению, то получаем новое доказанное утверждение. Утверждение, которое таким образом может быть выведено из аксиом, называют теоремой данной теории. Утверждение считается имеющим смысл, если оно есть либо аксиома, либо теорема данной теории. В отличие от экстенсионального подхода при интенсиональном подходе априори не определяется никакой универсум, и при таком подходе к определению семантики "иметь смысл" означает "быть теоремой или аксиомой данной теории".

Вернемся к рассмотренному выше языку арифметических выражений. Зададим его в виде формальной теории следующего вида.

Зададим язык арифметических выражений следующим образом:

1) аксиома — любой атом

$$a_1, \ldots, a_n;$$

2) правила вывода:

$$e_1, e_2 \Rightarrow (e_1+e_2), \quad (1)$$

$$e_1, e_2 \Rightarrow (e_1*e_2), \quad (2)$$

Построим доказательство приведенного ранее арифметического выражения:

a,b,c,d,e,g аксиомы

(e+g) правило вывода (1);

(d∗(e+g))	правило вывода (2);
(c+(d∗(e+g)))	правило вывода (1);
(b∗(c+(d∗(e+g))))	правило вывода (2);
(a+(b∗(c+(d∗(e+g)))))	правило вывода (1).

Нетрудно видеть, что мы определили тот же язык другим способом. Множество доказуемых утверждений совпадает здесь с множеством всех утверждений, и, таким образом, семантика в данном случае совпала с синтаксисом: утверждение имеет смысл тогда и только тогда, когда может быть доказано, т.е. тогда и только тогда, когда является арифметическим выражением, порождаемым приведенной выше грамматикой.

В этой связи полезно заметить, что и правила вывода формальной теории следует трактовать как просто "**правила вывода**" (в полной аналогии с порождающими грамматиками), согласно которым разрешается от определенных цепочек (в левой части правила) переходить к новым цепочкам (в правой его части).

Тогда по правилам вывода можно получать из цепочек, не обязательно теорем или аксиом, какие-то другие цепочки, т.е. строить выводы, не являющиеся доказательствами. Это опять-таки аналогично грамматикам, в которых применение продукций, вообще говоря, не обязано быть выводом из аксиомы.

Такая широкая и чисто синтаксическая трактовка правил вывода позволяет в теории формальных систем и в связанной с нею теории доказательств строить выводы из "гипотез" — цепочек, которые предполагаются доказанными. Если затем удается действительно доказать их, то построенное первоначально "относительное" доказательство превратится в "абсолютное" (т.е. начинающееся с аксиом).

Построив аксиоматическую семантику языка, мы можем на этой базе определить семантику уже экстенсионально, положив, что экстенсионал утверждения есть множество всех его доказательств (если утверждение не имеет доказательства, то его экстенсионал считается не определенным — формально мы включаем тогда в предметную область специальный "неопределенный элемент" с тем, как уже отмечалось, чтобы можно было вводимую семантическую функцию считать отображением).

5.3 Семантика простейшего языка программирования.

В заключение рассмотрим экстенсиональное определение семантики простейшего языка программирования, который будем называть MILAN

(MIniLANguage). Синтаксис языка определим посредством форм Бэкуса-Наура.

Определение семантики языка программирования MILAN (MIniLANguage).

(**программа**) ::= (последовательность_операторов)

(**последовательность_операторов**) ::= (оператор) | (оператор) последовательность_операторов)

(**оператор**) ::= (присваивание) | (условный лереход) | (цикл)

(**присваивание**) ::= (переменное) := (терм)

(**условный_переход**) ::= if (условие) then(последовательность_операторов)

else (последовательность-операторов) | if (условие) then последовательность_операторов)

(**цикл**) ::= while (условие) do (последовательность_операторов) end

(переменное) ::= z_i | ... | z_n

(**терм**) ::= (функциональный_символ) ((последовательность_термов)) | (переменное) | (константа)

(**последовательность_термов**) ::= (терм) | (терм) ((последовательность_термов))

(**функциональный_символ**) ::= f_1| … |f_m
(**константа**) ::= c_1|… |c_k
(**условие**) ::= (предикатный_символ) ((последовательность_термов))
(**предикатный_символ**) ::= p_1| … |p_s

Заданная таким образом КС-грамматика определяет так называемый абстрактный синтаксис MILANa: мы игнорируем некоторые синтаксические детали, такие, как слова begin, end, обрамляющие последовательность операторов, а также то, что операторы разделяются точкой с запятой, что между термами в последовательности термов ставится запятая и т.п. Мы также не уточняем структуру переменных (идентификаторов), констант, функциональных и предикатных символов, считая, что эти нетерминалы "пробегают" каждый свой алфавит.

Чтобы формально описать экстенсиональную семантику введенного посредством написанных выше синтаксических правил языка, нужно определить универсум.

Различают:

1) **денотационную** семантику;

2) **операциональную** семантику;

3) **трансформационную** семантику (семантику смешанных вычислений).

Мы будем рассмотривать только денотационную семантику языка, подобного MILANy, универсум которой определяется как множество преобразователей состояний памяти.

Состояние памяти *— это произвольное* ***отображение множества переменных*** **I** *языка программирования в* ***множество*** **D** ***данных (значений)***.

В зависимости от того, какой универсум рассматривается, в рамках экстенсионального подхода различают:

В денотационной семантике языка, подобного MILANy, универсум определяется как множество преобразователей состояний памяти. К построению этого множества и переходим.

Определение. Состояние памяти — это произвольное отображение множества переменных I языка программирования в множество D данных (значений).

Т.о. будем отождествлять память с множеством переменных (идентификаторов, "ячеек", имен) языка программирования I

σ: I→D,

сопоставляющее каждой переменной значение, принадлежащее некоторому множеству значений (данных) D.

В множество D включен также неопределенный элемент **O** (неопределенное значение).

Множество всех состояний памяти обозначим Σ.

В силу сформулированного определения мы отождествляем память с множеством переменных (идентификаторов, "ячеек", имен) данного языка программирования (конкретно — языка MILAN), а состояние памяти σ есть отображение вида σ: I→D, сопоставляющее каждой переменной значение, принадлежащее некоторому множеству значений (данных) D. Последнее можно рассматривать как объединение некоторого семейства множеств, служащих носителями многосортной алгебры данных языка. Их можно отождествить с типами данных, используемыми в рассматриваемом языке: числами, массивами, строками, стуктурами и т.п. В множество D включен также неопределенный элемент O (неопределенное значение). Множество всех состояний памяти обозначим Σ.

Введем понятие преобразователя состояний памяти.

Преобразователь состояний памяти — это произвольное отображение множества Σ состояний памяти в себя.

Множество всех преобразователей состояний памяти обозначим (**Σ → Σ**).

На множестве Σ определяется структура индуктивного упорядоченного множества.

Область определенности состояния памяти σ - это множество, обозначаемое D(σ), всех переменных x, таких, что σ(x) ≠ **O**.

Тогда положим для двух произвольных состояний σ и ρ

$$\Sigma \leqslant \rho \Leftrightarrow (D(\sigma) \subseteq D(\rho)) \,\&\, ((\forall x \in D(\sigma))\ (\sigma(x) = \rho(x))).$$

Множество Σ имеет по отношению ⩽ наименьший элемент, а именно такое состояние **O**, что (∀x ∈ I) (**O**(x) = **O**).

Его называют **всюду неопределенным состоянием памяти**; ясно, что D(O) = ∅.

Кроме того, для любой неубывающей последовательности состояний памяти

$$\sigma_1 \leqslant \sigma_2 \leqslant \ldots \leqslant \sigma_n \leqslant \ldots$$

состояние σ_0, такое, что $\sigma_0(x) = \sigma_n(x) \Leftrightarrow x \in D(\sigma_n)$, есть точная верхняя грань этой последовательности, причем $D(\sigma_0) = \bigcup_{n \geqslant 1} D(\sigma_n)$.

Таким образом, множество состояний памяти Σ, снабженное отношением порядка ⩽, **является индуктивным упорядоченным множеством**.

Из этого следует, что и множество преобразователей состояний (Σ → Σ) есть индуктивное упорядоченное множество. В этом множестве отношение порядка ⩽ определяется условием

$$\mathbf{f \leqslant g \Leftrightarrow (\forall \sigma \in \Sigma)\ (f(\sigma) \leqslant g(\sigma))},$$

наименьшим элементом является стирающий преобразователь 0, такой, что 0(σ)=**O** для всякого σ∈Σ, а точной верхней гранью неубывающей последовательности преобразователей состояний

$\mathbf{f_1 \leqslant f_2 \leqslant \ldots \leqslant f_n \leqslant \ldots}$ является преобразователь f_0, определяемый следующим образом:

$\mathbf{(\forall \sigma \in \Sigma)\ (\forall x \in I)(f_0(\sigma)(x) = f_n\ (\sigma)(x) \Leftrightarrow x \in D(f_n\ (\sigma)))}$, причем $\mathbf{D(f_0(\sigma)) = \bigcup_{n \geqslant 1} D(f_n(\sigma))}$.

Более того, оказывается, что композиция преобразователей состояний (как отображений) непрерывна в смысле сохранения точных верхних граней, точнее, **для любой** неубывающей последовательности

преобразователей состояний $f_1 \leqslant f_2 \leqslant \ldots \leqslant f_n \leqslant \ldots$ и произвольного преобразователя состояний g имеем…

Для любой неубывающей последовательности преобразователей состояний $f_1 \leqslant f_2 \leqslant \ldots \leqslant f_n \leqslant \ldots$ и произвольного преобразователя состояний g имеем

$$\mathbf{g \circ sup f_n = sup(g \circ f_n),\ sup f_n \circ g = sup(f_n \circ g)}.$$

Докажем 1-е из этих равенств. Для произвольного $\sigma \in \Sigma$ имеем

$$\mathbf{g \circ sup f_n(\sigma) = sup f_n\,(g(\sigma))}.$$

Тогда для всякого

$$\mathbf{x \in D(g \circ f_n(\sigma))}$$

имеем

$$\mathbf{g \circ sup f_n(\sigma)(x) = sup f_n(g(\sigma))(x) = f_n\,(g(\sigma))(x) = g \circ f_n(\sigma)(x),}$$

т.е. поскольку, как можно показать, композиция монотонна в смысле определения, данного ранее, то

$$\mathbf{D(g \circ sup f_n(\sigma)) = \bigcup_{n \geqslant 1} D(g \circ f_n)(\sigma)}$$

и

$$\mathbf{g \circ sup f_n = sup(g \circ f_n).}$$

5.4 Денотационная семантика языка MILAN.

Имея в виду все рассмотренные выше свойства множества $(\Sigma \to \Sigma)$, определим денотационную семантику языка MILAN.

Семантическая функция есть отображение $[\![\circ]\!]$ множества фраз в множество $(\Sigma \to \Sigma)$ преобразователей состояний памяти.

Семантика оператора присваивания:

$$[\![x := t]\!](\sigma)(y) = \begin{cases} \sigma(y), & \text{if } y \neq x; \\ t^{\sigma}, & \text{if } y = x. \end{cases}$$

где через t^{σ} обозначено значение терма t в состоянии σ, равное 0, если хотя бы для одного переменного, входящего в терм, его значение не определено, и равное значению терма при подстановке на место каждого его переменного x его значения σ(x).

Суть записанного выше семантического правила очень проста: оператору присваивания x := t сопоставляется преобразователь состояний, меняющий состояние таким образом, что все переменные, кроме x, сохраняют свои значения, а новое значение переменного x есть значение правой части оператора присваивания (терма t) в состоянии σ.

Семантика оператора условного перехода

$$[\![\text{if } p(t_1, \dots, t_n) \text{ then } S_1 \text{ else } S_2]\!] = \begin{cases} [\![S_1]\!](\sigma), & \text{if } p(t_1^{\sigma}, \dots, t_n^{\sigma}) = 1; \\ [\![S_2]\!](\sigma), & \text{if } p(t_1^{\sigma}, \dots, t_n^{\sigma}) = 0. \end{cases}$$

Это правило написано в предположении, что значения всех термов в состоянии σ определены. В противном случае оператору условного перехода сопоставляется стирающий преобразователь ∘. Подобное же соглашение принимается и далее в аналогичных ситуациях.

Семантика условного перехода без else-альтернативы записывается аналогично, но при условии ложности предиката p в состоянии σ берется тождественный преобразователь состояний ID.

Семантика цикла

$$[\![\text{while } p(t_1, \dots, t_n) \text{ do S end}]\!](\sigma) = \begin{cases} [\![S]\!] \circ [\![\text{while } p(t_1, \dots, t_n) \text{ do S end}]\!](\sigma), & \text{if } p(t_1^{\sigma}, \dots, t_n^{\sigma}) = 1; \\ ID(\sigma), & \text{if } p(t_1^{\sigma}, \dots, t_n^{\sigma}) = 0. \end{cases}$$

Таким образом, семантика цикла определяется как решение уравнения. Это определение корректно, так как правая часть уравнения есть непрерывное отображение множества $(\Sigma \to \Sigma)$ в себя (что следует из доказанной выше непрерывности композиции преобразователей состояний и очевидной непрерывности тождественного отображения).

Семантика последовательности операторов:

$$[\![S_1; S_2]\!] = [\![S_1]\!] \circ [\![S_2]\!].$$

Определенная таким образом формальная семантика языка MILAN становится базой для семантического анализа программ, т.е. для строгого математического доказательства утверждений о программах. Этот анализ, проводимый технологически после синтаксического анализа программы и получения ее дерева вывода, основан на результате, известном под названием принципа индукции по не неподвижной точке.