Автоматы и формальные языки

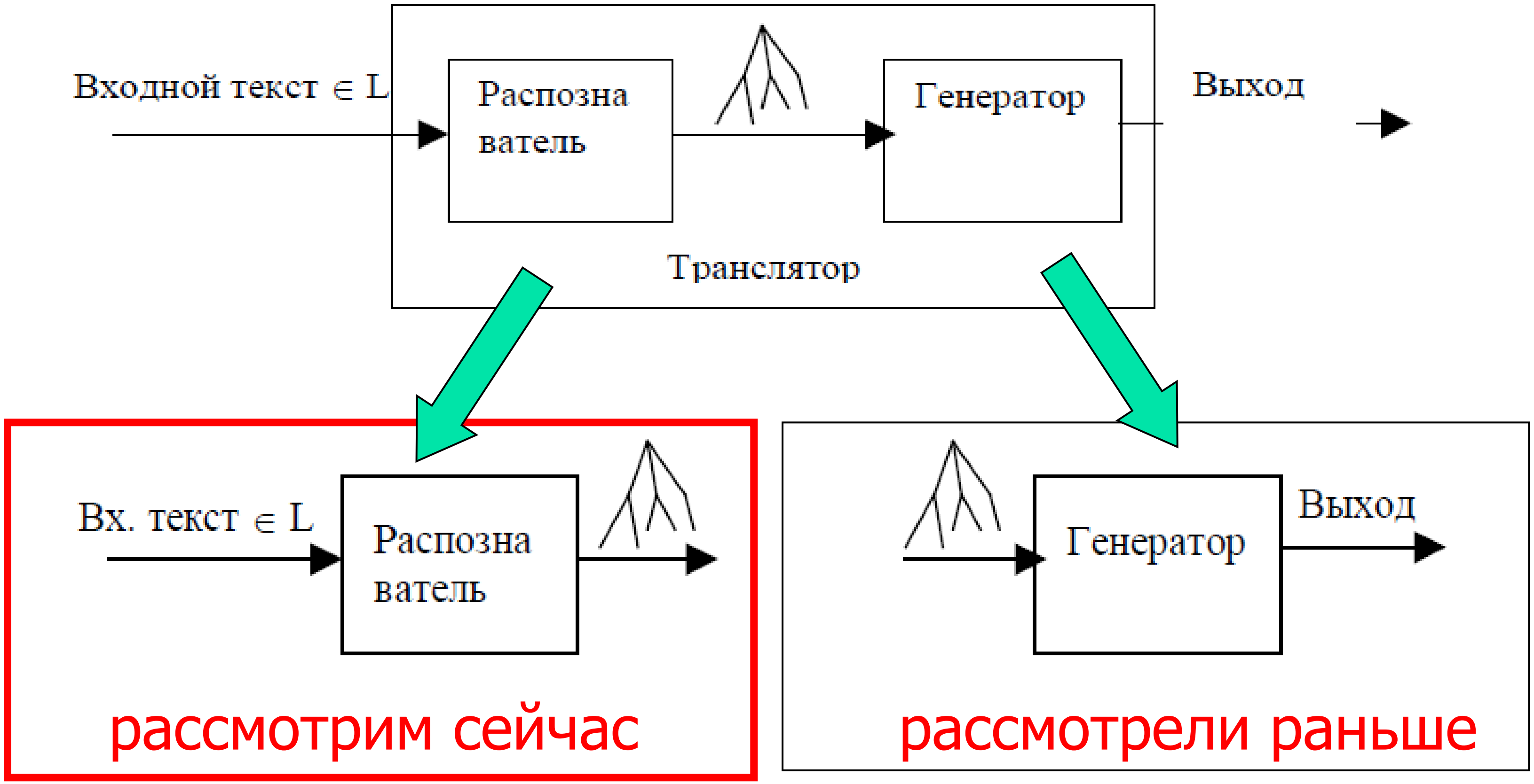
Карпов Юрий Глебович
профессор, д.т.н., зав.кафедрой
“Распределенные вычисления и компьютерные сети”
Санкт-Петербургского Политехнического университета

karpov@dcn.infos.ru

Структура курса

- Конечные автоматы-распознаватели – 4 л
- Порождающие грамматики Хомского – 3 л
- Атрибутные трансляции и двусмысленные КС-грамматики – 2 л
- Распознаватели КС-языков и трансляция – 6 л
  - Лекция 10. s-грамматики, LL(k)-грамматики, грамматики рекурсивного спуска
  - Лекция 11. Построение транслятора языка Милан методом рекурсивного спуска
  - Лекция 12. Грамматики предшествования, LR(k)-грамматики
  - Лекция 13. SLR(k) и LALR(k)-грамматики.
  - Лекция 14. Компиляторы компиляторов. Yacc и Bison.
  - Лекция 15. Грамматики Кока-Янгера-Касами и Эрли
- Дополнительные лекции - 2 л

Задачи распознавателя и генератора

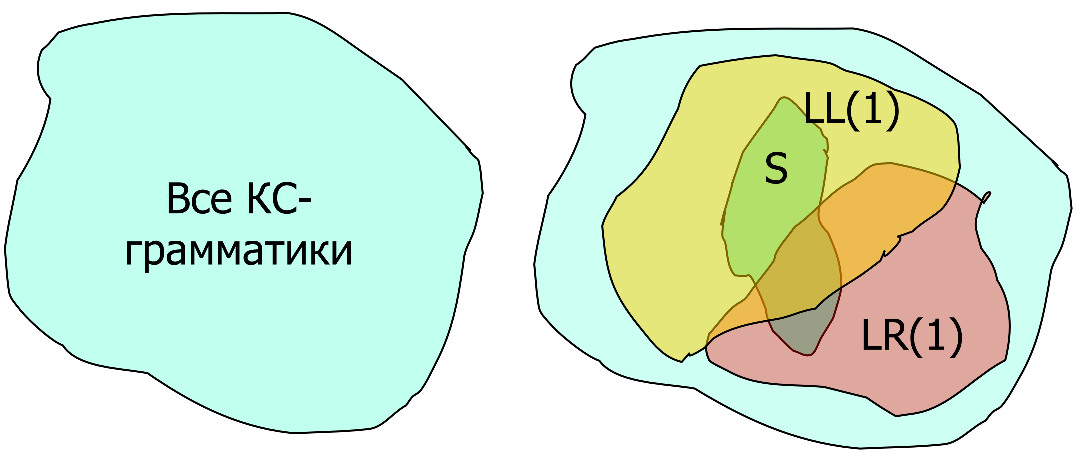


Сегодня рассмотрим, как построить дерево вывода любой входной цепочки языка, порождаемого заданной грамматикой

Синтаксический анализ - одна из наиболее глубоко проработанных и понятных ветвей Computer Science

- Для КС-грамматик синтаксический анализ - это восстановление дерева вывода данной цепочки в данной грамматике, фактически, это алгоритм проверки принадлежности входной цепочки языку, заданному КС-грамматикой

Существуют “универсальные”алгоритмы СА, работающие для любой КС-грамматики, но они неэффективны: они должны уметь строить несколько деревьев



Подклассы КС-грамматик
- s-грамматики
- LL(k)-грамматики,
- Грамматики рекурсивного спуска
- Грамматики предшествования
- LR(k)-грамматики,
- LALR(k)-грамматики,
- ...

Возможность применения эффективных алгоритмов СА зависит от вида КС-грамматики

Функция грамматики языка

- Грамматика – конечный формализм определения бесконечного числа предложений языка
- Но функции грамматики не только в этом: на основе грамматики выполняются и синтаксический анализ, и семантические вычисления
- Формальный язык может иметь практическую ценность только в том случае, если для него можно построить эффективный транслятор, а эффективность транслятора зависит от вида грамматики
- Для языка нужно ПОДБИРАТЬ грамматику, позволяющую выполнить эффективную трансляцию (эффективный синтаксический анализ и эффективные семантические вычисления)
- При создании языка в грамматику языка (и в язык!) включаются правила с необходимыми разделителями, со структурой, которая позволяет эффективно выполнить синтаксический анализ и семантические вычисления

Преобразования грамматик, сохраняющие эквивалентность

Преобразования КС-грамматик, сохраняющие эквивалентность

- Две грамматики эквивалентны, если они порождают один и тот же язык
- Проблема проверки эквивалентности двух КС-грамматик в общем случае неразрешима. Однако часто удается преобразовать грамматику к виду (удобному для применения того или иного метода анализа), не изменяя язык, порождаемый этой грамматикой

Это, фактически, задача приведения КС-грамматики к "хорошему", "чистому" виду: например:

выбрасывание неиспользуемых правил;
выбрасывание правил вида $A \rightarrow A$;
выбрасывание нетерминалов, которые вообще не встречаются в выводах

Применение этих алгоритмов к грамматике называют "гигиеной" грамматик

Приведение КС-грамматик
(выбрасывание правил, не влияющих на выводы)

Нетерминал A полезный, iff

$$S \Rightarrow^* \alpha A \beta \Rightarrow^* \gamma \qquad \gamma \in T^*$$

его можно "вывести" из S

из него выводится произвольная терминальная цепочка

G:: $S \to aSb$
$S \to cA$
$S \to aD$
$A \to aAbB$
$A \to ad$
$B \to baa$
$B \to bA$
$C \to abS$ ⟶ нетерминал C не выводится из S
$A \to A$ ⟶ бессмысленное правило
$D \to bDcD$ ⟶ нетерминал D выводится из начального символа, но из него нельзя вывести терминальную строку

Все правила, содержащие бесполезные нетерминалы, могут быть удалены из грамматики Порождаемый язык не изменится

- Нетерминал A *достижимый*, если существует вывод $S \Rightarrow^* \alpha A \beta$.
  Нетерминал A *продуктивный*, если из него можно вывести терминальную цепочку, т.е. существует вывод $A \Rightarrow^* \gamma$, $\gamma \in T^*$
- Полезные нетерминалы: *достижимые* И *продуктивные*

- Приведение грамматик: найти все бесполезные нетерминалы и выбросить из грамматики все правила, в которых они встречаются

Построение множества достижимых нетерминалов

- Для построения множества достижимых нетерминалов последовательно строятся множества $M_0, M_1, \dots M_k, \dots$
- Начальное множество M_0 достижимых нетерминалов равно {S}. В следующее множество M_{k+1} добавляются нетерминалы B, такие, что в грамматике существует правило $A \Rightarrow \alpha B \beta$ для какого-нибудь A из предыдущего множества M_k

$M_0=\{S\}$, // любой вывод начинается с S

$M_{k+1}= M_k \cup \{B \mid \exists A{\in}M_k: A \Rightarrow \alpha B \beta \}$ // добавляем те, которые выводятся

Пример.
$S \to SbAc \mid dA \mid d$
$A \to AcC \mid abc \mid dAE$
$B \to ScC \mid dB$
$C \to cC \mid DdAS$
$D \to cE \mid CdA$
$E \to Ac \mid Dd$
$F \to cC \mid a \mid dAE$
$G \to AcC \mid bc \mid dA$

$M_0=\{S\};$
$M_1=\{S, A\};$
$M_2=\{S, A, C, E\};$
$M_3 =\{S, A, C, E, D\};$
$M_4=M_3$

$$M_\infty =\{S, A, C, D, E\}$$

Пример.
$S \to SbAc \mid dA \mid d$
$A \to AcC \mid abc \mid dAE$
$\cancel{B \to ScC \mid dB}$
$C \to cC \mid DdAS$
$D \to cE \mid CdA$
$E \to Ac \mid Dd$
$\cancel{F \to cC \mid a \mid dAE}$
$\cancel{G \to AcC \mid bc \mid dA}$

Бесполезные правила выбрасываем

Построение множества продуктивных нетерминалов
(из которых могут быть порождены терминальные цепочки)

- Для построения множества продуктивных нетерминальных символов последовательно строятся множества V_0, … V_k … ,
- V_0 включает такие нетерминалы, из которых за одни шаг выводятся терминальные цепочки
- $V_{k+1} = V_k \cup \{B \mid (\exists \alpha \in V^*): B \Rightarrow \alpha\}$ - в следующее множество V_{k+1} добавляются нетерминалы B такие, что в грамматике существует правило $B \Rightarrow \alpha$, где α включает терминалы и нетерминалы только из предыдущего множества V_k

Грамматика G

$S \to SbAc \mid dA \mid d$
$A \to AcC \mid abc \mid dAE$
$B \to ScC \mid dB$
$C \to cC \mid DdAS$
$D \to cE \mid CdA$
$E \to Ac \mid Dd$
$F \to cC \mid a \mid dAE$
$G \to AcC \mid bc \mid dA$

$V_0=\{S, A, F, G\}$;

$V_1=\{S, A, F, G, E\}$;

$V_2=\{S, A, F, G, E, D\}$; $V_3=V_2$

$M_\infty = \{S, A, C, E, D\}$

$V_\infty = \{S, A, F, G, E, D\}$

Берем пересечение M и V

$M_\infty \cap V_\infty = \{S, A, D, E\}$

В грамматике оставляем только те правила, в которые входят нетерминалы только из пересечения

Грамматика G′

$S \to SbAc \mid dA \mid d$
$A \to abc \mid dAE$
$D \to cE$
$E \to Ac \mid Dd$

Построили приведенную грамматику G′

Пример

- Устранить бесполезные нетерминалы в грамматике:

  $S \to aC \mid Ac \mid bDe$

  $A \to cAB$

  $B \to b \mid dB$

  $C \to cCB \mid a$

  $E \to cE \mid BdA \mid d$

$M_0=\{S\}$;
$M_1=\{S, C, A, D\}$;
$M_2=\{S, C, A, D, B\}$; $M_3= M_2$

$V_0=\{C, B, E\}$;
$V_1=\{S, C, B, E\}$; $V_2= V_1$

- Решение
  - Достижимые терминальные символы: {S, A, B, C, D}
  - Продуктивные терминальные символы: {B, C, E, S}
  - Полезные терминальные символы (пересечение множеств достижимых и продуктивных): {S, B, C}
- Приведенная грамматика:

  $S \to aC$

  $B \to b \mid dB$

  $C \to cCB \mid a$

Проблема пустоты КС-языка разрешима: язык, порождаемый КС-грамматикой НЕпуст, если и только если S является продуктивным символом

ε - свободные и неукорачивающие КС-грамматики

- **Определение.** КС-грамматика называется неукорачивающей, если она не включает продукций вида $A \to \varepsilon$
- Любой шаг вывода в неукорачивающей грамматике не может уменьшить длину выводимой цепочки – отсюда и ее название.
- КС-грамматика называется ε - свободной, если она неукорачивающая или в ней существует ровно одна продукция вида $S \to \varepsilon$, где S – начальный нетерминал, и S не встречается в правой части ни одной из продукций

Приведение к ε-свободной грамматике

- **Теорема.** По любой КС-грамматике может быть построена эквивалентная ε - свободная КС- грамматика. По любой КС-грамматике, порождающей язык, не включающий пустую цепочку, может быть построена эквивалентная неукорачивающая грамматика.
  - **Доказательство.** Для любого правила $A \to \varepsilon$ скопируем (повторим) все продукции, включающие A в правой части, с выбрасыванием символа A из правых частей этих копий, причем если A входит несколько раз в правую часть продукции, то такое выбрасывание произведем во всех возможных сочетаниях. Далее, продукцию $A \to \varepsilon$ выбросим из множества R продукций грамматики.
  - Если S - начальный нетерминал, и $S \to \varepsilon$ – среди продукций, то выбираем новый начальный нетерминал S′, и к продукциям грамматики добавляем две новых: $S' \to \varepsilon$ и $S' \to S$

- Пример. Привести к неукорачивающему виду грамматику:

  $S \to cA \mid \varepsilon$

  $A \to cA \mid bA \mid \varepsilon$

- Решение.

  $S' \to S \mid \varepsilon$

  $S \to cA \mid c$

  $A \to cA \mid bA \mid c \mid b$

ε – свободные языки

- **Определение.** Язык называется ε - свободным, если он не включает пустой цепочки
- По любой КС-грамматике можно проверить, является ли порождаемый ею язык ε - свободным
  - Для этого нужно проверить, есть ли в эквивалентной ε - свободной грамматике продукция $S \to \varepsilon$.
- Любую грамматику можно преобразовать в неукорачивающую:
  - добавим в грамматику новый нетерминал S' и продукцию $S' \to S\$$
  - применим алгоритм приведения грамматики к ε - свободному виду

Следствие: Проблема пустоты (содержит ли порождаемый грамматикой язык пустую цепочку?) для КС грамматик разрешима

В некоторых случаях для упрощения удобно к грамматике добавлять правило $S' \to \# S \#$

Ясно, цепочки языка, порождаемого новой грамматикой будут иметь вид $\#\alpha\#$, где α - цепочка прежнего языка

Сингулярные продукции (вида $A \rightarrow B$)

- Эффективный алгоритм, позволяющий по любой ε - свободной КС-грамматике построить эквивалентную КС-грамматику без сингулярных продукций:
  - Включим в множество R продукций грамматики G вместо каждой сингулярной продукции вида $A \rightarrow B$ все продукции $A \rightarrow \beta$ такие, что продукция $B \rightarrow \beta \in R$ и несингулярна

- **Пример.** Для грамматики:
  1. $S \rightarrow BA$
  2. $A \rightarrow C \mid ac$
  3. $B \rightarrow b$
  4. $C \rightarrow A$

эквивалентная грамматика без сингулярных продукций имеет вид:
  1. $S \rightarrow BA$
  2. $A \rightarrow ac$
  3. $B \rightarrow b$
  4. $C \rightarrow ac$

В результате приведения этой грамматики последняя продукция будет выброшена. Эквивалентная приведенная грамматика:
  1. $S \rightarrow BA$
  2. $A \rightarrow ac$
  3. $B \rightarrow b$

Сингулярные продукции: осторожно!

- В языках программирования сингулярные продукции используются достаточно часто, и их выбрасывание может нарушить очевидность структуры продукций. Исключение составляет специальный случай сингулярных продукций, которые приводят к циклу вида $A \Rightarrow^+ A$.
- ***Определение.*** КС-грамматика, в которой не существует вывода $A \Rightarrow^+ A$, называется ациклической.
  Любую КС-грамматику можно привести к эквивалентному ациклическому виду.

Действительно, построим эквивалентную ε - свободную грамматику, и затем приведем ее к эквивалентному виду без сингулярных продукций.

Левая рекурсия

- ***Определение.*** *КС-грамматика называется грамматикой с левой рекурсией, если в ней существует вывод $A \Rightarrow^* A\beta$.*
  - Леворекурсивные грамматики не всегда удобны. В частности, нисходящие методы синтаксического анализа, восстанавливающие дерево вывода от его корня, не могут быть применены для грамматик с левой рекурсией. Существует алгоритм, позволяющий привести любую КС-грамматику к эквивалентному виду без левой рекурсии
- Рассмотрим случай "**прямой левой рекурсии**", когда грамматика имеет продукции вида $A \rightarrow A\alpha$
- Очевидно, что A будет "полезным" символом, только если в грамматике есть продукции вида $A \rightarrow \beta$, т.е. грамматика содержит, как минимум, пару продукций
  $A \rightarrow A\alpha \mid \beta$
  Многократное применение правила $A \rightarrow A\alpha$ даст $A \Rightarrow^* A\alpha^*$, и последняя замена A на β даст $A \Rightarrow^* \beta\alpha^*$

$$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha \Rightarrow^* A\alpha^* \Rightarrow \beta\alpha^*$$

- Вывод: пара продукций
  $A \rightarrow A\alpha \mid \beta$ // из A выводятся только цепочки вида $\beta\alpha\alpha\alpha\alpha$ ...
  может быть заменена следующими нелеворекурсивными продукциями:
  $A \rightarrow \beta X$
  $X \rightarrow \alpha X \mid \varepsilon$

Пример

- Классическая грамматика арифметических выражений:
  $E \to E+T \mid T$
  - Используя алгоритм, преобразуем эти правила с прямой левой рекурсией к следующему эквивалентному виду:
    $E \to TE'$
    $E' \to +TE' \mid \varepsilon$
- Этот метод легко обобщается на случай, когда нетерминал имеет несколько альтернатив с прямой левой рекурсией. Более сложный общий случай избавления от непрямой левой рекурсии требует чуть более сложного алгоритма

В БНФ-нотации грамматика арифметических выражений без левой рекурсии:

$E \to E+T \mid T$
$T \to T*P \mid P$
$P \to i$

→

E ::= T {+T}
T ::= P {* P}
P ::= i

<выражение>- это **<терм>** с идущими за ним произвольным числом **+<терм>**

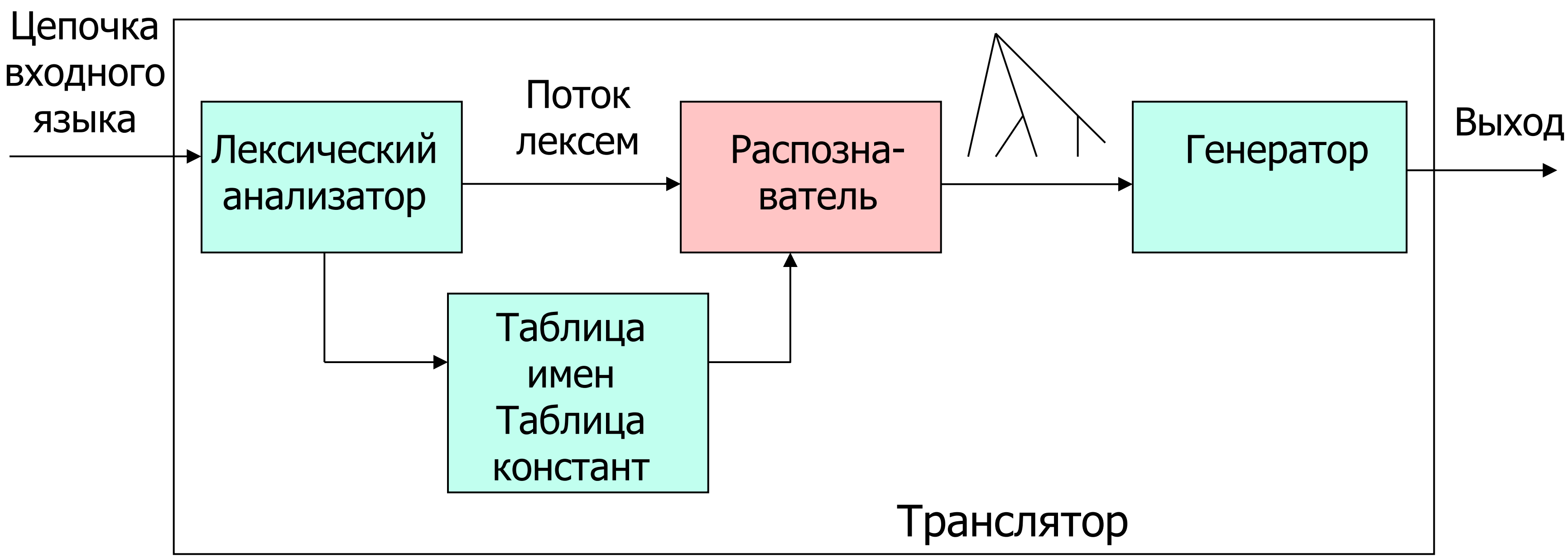
<терм>- это **<первичное>** с идущими за ним произвольным числом ***<первичное>**

'+' можно понимать, как <u>addop</u>, '*' – как <u>mulop</u>

Постановка задачи синтаксического анализа

Синтаксический анализ КС-языков

- Для КС-грамматик синтаксический анализ - это восстановление дерева вывода данной цепочки в данной грамматике



Наша проблема – проблема построения алгоритма распознавания: как воссоздать вывод произвольной цепочки в заданной грамматике?

Синтаксический анализ – одна из наиболее проработанных и понятных ветвей Computer Science, и это - очень красивая ветвь

Задача блока синтаксического анализа (СА)

- Рассмотрим пример КС-грамматики.

  $G_{4.5}$: 1. $S \rightarrow abScB$
  2. $S \rightarrow bA$
  3. $A \rightarrow ab$
  4. $A \rightarrow cBA$
  5. $B \rightarrow bBc$
  6. $B \rightarrow c$

  Пусть в этой грамматике есть вывод некоторой терминальной цепочки:

  $$S \Rightarrow abScB \Rightarrow abbAcB \Rightarrow abbcBAcB \Rightarrow abbccAcB \Rightarrow abbccabcB \Rightarrow abbccabcc$$

- Задача блока синтаксического анализа обратна: по заданной терминальной цепочке восстановить вывод ее из начального символа грамматики:

  $$S \Rightarrow \ \ldots \quad ? \ ? \ ? \ ? \quad \ldots \ \Rightarrow abbccabcc$$

- Синтаксический анализ можно понимать и как восстановление синтаксического дерева заданной входной цепочки. Для этого дерева известны листья – (входная цепочка) и корень (начальный символ грамматики)

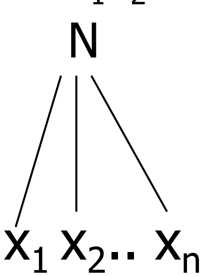
Нисходящие алгоритмы синтаксического анализа

$S \Rightarrow_1 abScB \Rightarrow_2 abbAcB \Rightarrow_4 abbcBAcB \Rightarrow_6 abbccAcB \Rightarrow_3 abbccabcB \Rightarrow_6 abbccabcc$

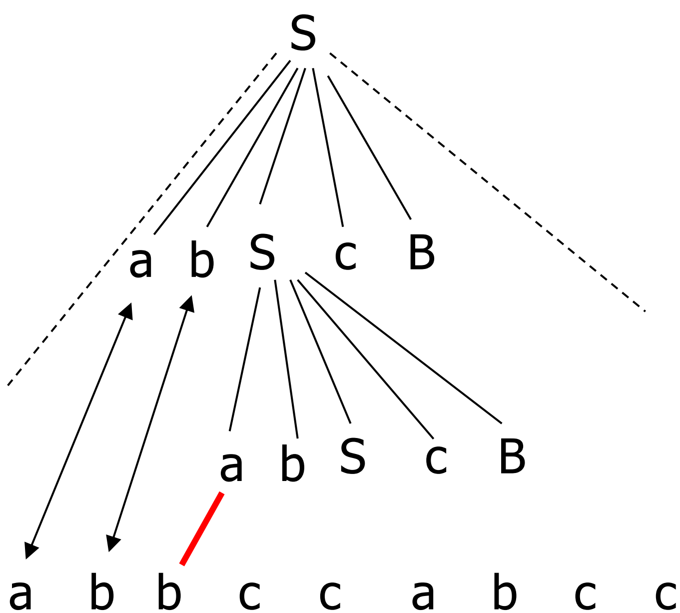
левый вывод

1. $S \to abScB$
2. $S \to bA$
3. $A \to ab$
4. $A \to cBA$
5. $B \to bBc$
6. $B \to c$

$N \to X_1X_2 \ldots X_n$



связка



нужно вернуться

Нисходящий синтаксический анализ работает так:

Начиная от начального символа дерева пытаемся найти, какой альтернативой (какой правой частью продукции) заменить очередной нетерминал, чтобы раскрыть новый узел синтаксического дерева. Восстанавливает ЛЕВЫЙ канонический вывод

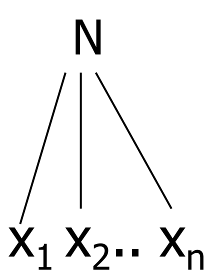
Повторяющимся вопросом здесь является следующий: какую альтернативу для текущего нетерминала нужно выбрать, чтобы из начального символа по текущей сентенциальной форме построить следующую сентенциальную форму вывода, чтобы получить входную цепочку?

Восходящие алгоритмы синтаксического анализа

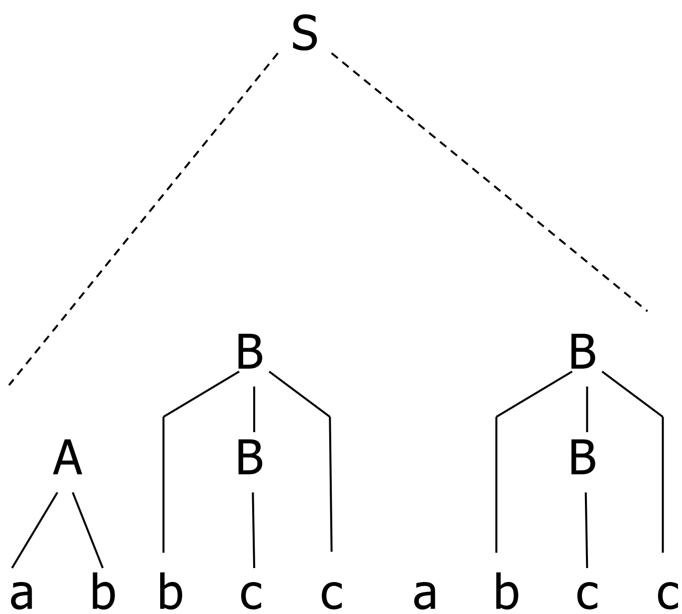
$S \Rightarrow_1 abScB \Rightarrow_2 abScc \Rightarrow_4 abbAcc \Rightarrow_6 abbcBAcc \Rightarrow_3 abbcBabcc \Rightarrow_6 abbccabcc$

правый вывод

1. $S \rightarrow abScB$
2. $S \rightarrow bA$
3. $A \rightarrow ab$
4. $A \rightarrow cBA$
5. $B \rightarrow bBc$
6. $B \rightarrow c$



связка



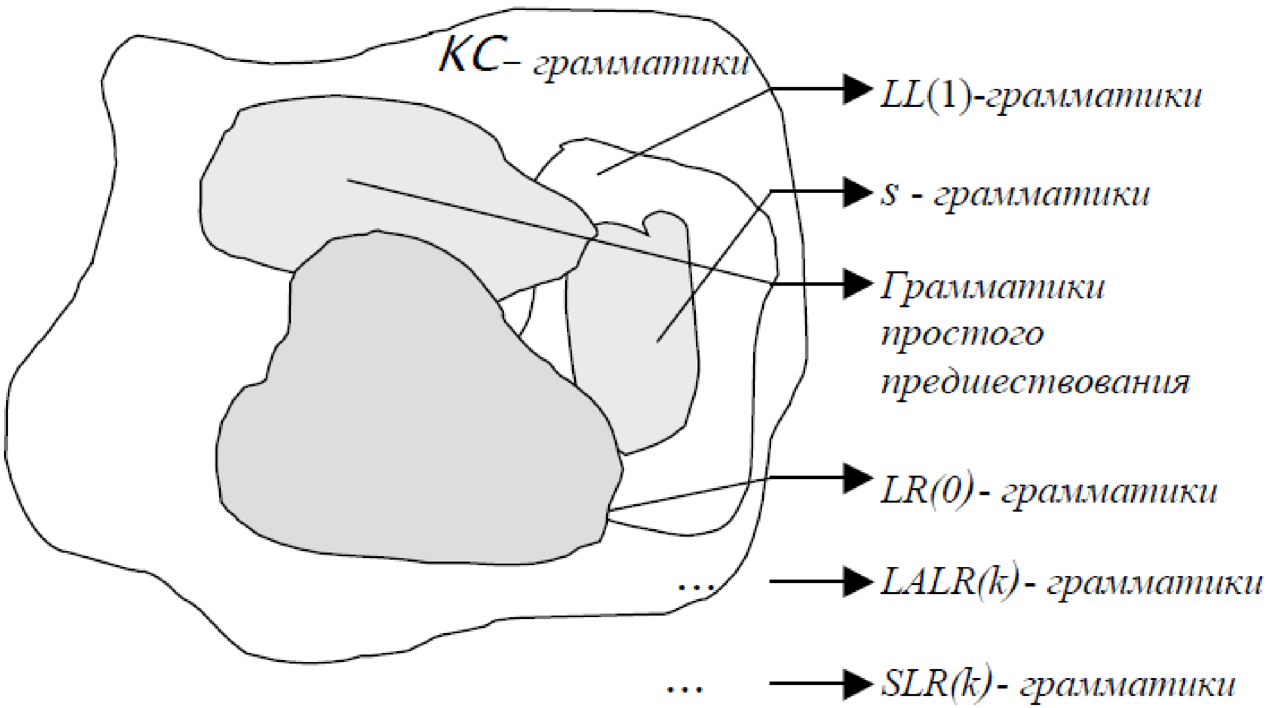
нужно вернуться

Восходящий синтаксический анализ работает так.

Начиная от терминальной строки листьев дерева пытаемся найти "связку" – правую часть продукции грамматики, которую нужно заменить нетерминалом (левой частью продукции), чтобы получить новый узел синтаксического дерева. Восстанавливает ПРАВЫЙ канонический вывод

Основным повторяющимся вопросом здесь является следующий: какую подстроку в сентенциальной форме нужно выбрать в качестве связки, и каким нетерминалом ее заменить с получением предыдущей сентенциальной формы вывода, чтобы свернуть всю цепочку к S?

Подклассы КС-грамматик, допускающие эффективные методы синтаксического анализа



Для всех КС-грамматик существуют общие неэффективные алгоритмы как нисходящие, так и восходящего синтаксического анализа (сложность n^3). Выделенные собственные подклассы – это те КС-грамматики, для которых были найдены различные по простоте и эффективности специфические алгоритмы нисходящего либо восходящего синтаксического анализа

Нисходящие методы синтаксического анализа

Нисходящий синтаксический анализ

$S \Rightarrow_1 abScB \Rightarrow_2 abbAcB \Rightarrow_4 abbcBAcB \Rightarrow_6 abbccAcB \Rightarrow_3 abbccabcB \Rightarrow_6 abbccabcc$

1. $S \to abScB$
2. $S \to bA$
3. $A \to ab$
4. $A \to cBA$
5. $B \to bBc$
6. $B \to c$

$S \Rightarrow \dots ??? \Rightarrow \dots abbccabcc$ КАК ВОССТАНОВИТЬ?

1. По какому правилу заменен S на I шаге, по 1 или 2?
   Можем гадать, а можем смотреть на входную цепочку abbccabcc

$S \Rightarrow_1 abSBc$

$S \Rightarrow_2 bA$

Что выберем? По первому символу анализируемой цепочки!! Выбираем $\Rightarrow_1$

$S \Rightarrow_1 abSBc \Rightarrow$ .... $\Rightarrow$ abbccabcc КАК ВОССТАНОВИТЬ ДАЛЬШЕ?

S	$\Rightarrow_1$	abSBc	$\Rightarrow_1$	ababScBBc ??	$\Rightarrow^*$	abbccabcc	эта подстановка для S?
S	$\Rightarrow_1$	abSBc	$\Rightarrow_2$	abbABc ??	$\Rightarrow^*$	abbccabcc	эта подстановка для S?

- Рассуждения о восстановлении вывода цепочки из начального символа эквивалентны рассуждениям о восстановлении синтаксического дерева: восстановление каждого шага вывода соответствует добавлению очередного узла в дерево, и обратно

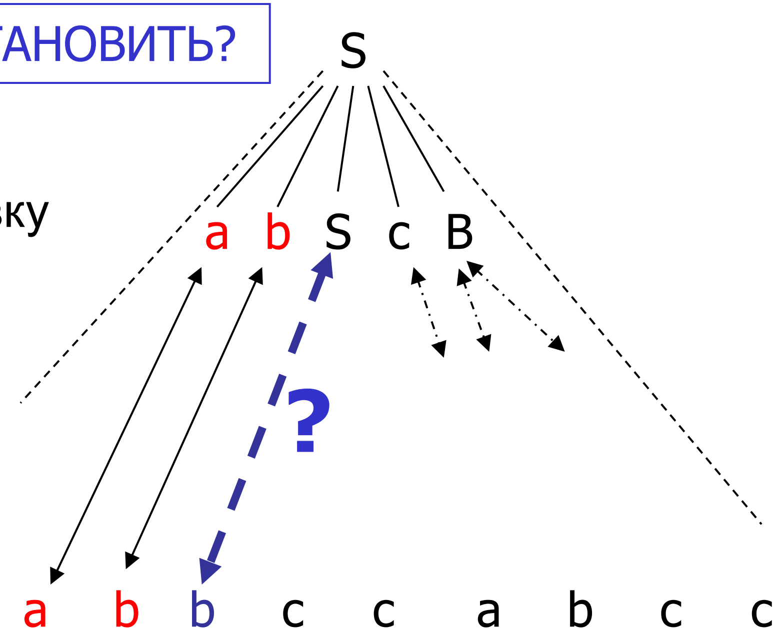
Нисходящий синтаксический анализ

$S \Rightarrow_1 abScB \Rightarrow_2 abbAcB \Rightarrow_4 abbcBAcB \Rightarrow_6 abbccAcB \Rightarrow_3 abbccabcB \Rightarrow_6 abbccabcc$

1. $S \to abScB$
2. $S \to bA$
3. $A \to ab$
4. $A \to cBA$
5. $B \to bBc$
6. $B \to c$

$S \Rightarrow^* abbccabcc$ КАК ВОССТАНОВИТЬ?

Пусть первую подстановку для S сделали



$S \Rightarrow_1 abScB$

$S \Rightarrow_2 bA$

1. По какому правилу теперь заменять S, по 1 или 2?

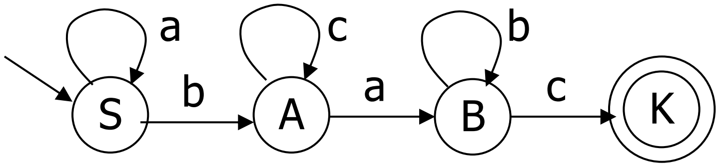
Можем гадать, а можем смотреть на остаток входной цепочки . . b c c a b c c

Что выберем? По очередному терминальному символу входной цепочки!!

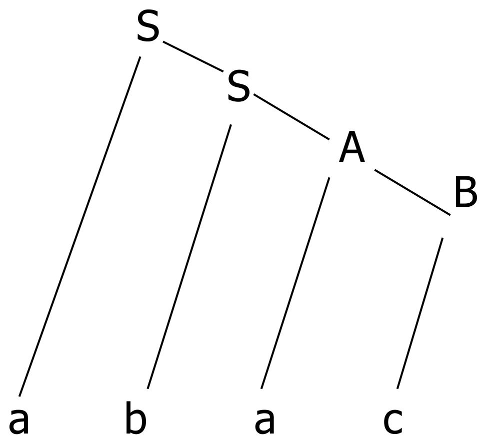
При выводе цепочки в КС-грамматиках если терминальный символ попал в начало сентенциальной формы, то он там и останется в процессе всего вывода

s-грамматики (simple grammars)

Автоматные грамматики и s-грамматики



1. S → aS
2. S → bA
3. A → aB
4. A → cA
5. B → bB
6. B → c



S⇒aS⇒abA⇒abaB ⇒abac

	a	b	c
S	S	A	ош
A	B	ош	A
B	ош	B	!

Таблица переходов

1. S → abScB
2. S → bA
3. A → ab
4. A → cBA
5. B → bBc
6. B → c

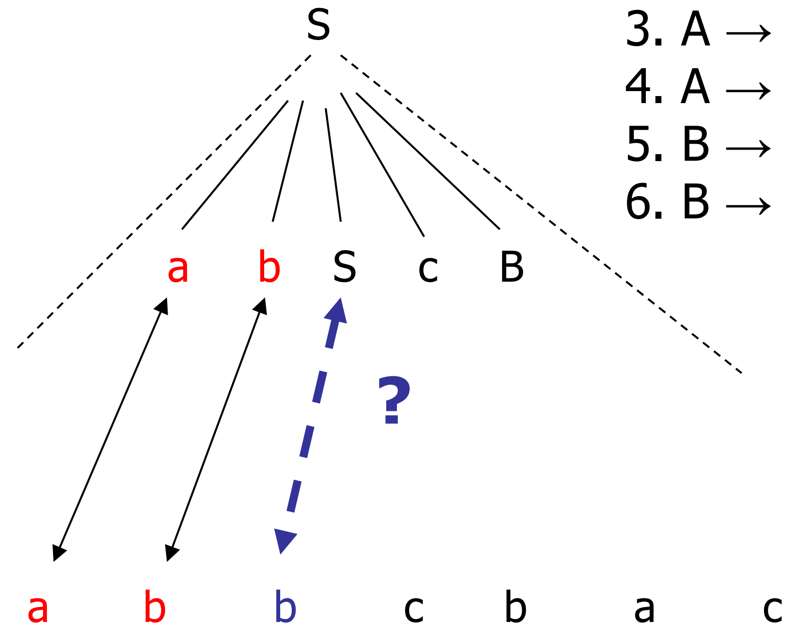


Таблица принятия решений

	a	b	c
S	abScB	bA	ош
A	ab	ош	cBA
B	ош	bBc	c
a	X	ош	ош
b	ош	X	ош
c	ош	ош	X

s-грамматики: пары <нетерминал, терминал>

- Автоматные языки удобны тем, что для них может быть построен детерминированный конечный автомат, реализующий эффективный (линейный) нисходящий алгоритм распознавания входной цепочки.
- По каждой паре: <*очередной (самый левый) нетерминал сентенциальной формы Q, очередной терминал входной строки*> распознаватель автоматного языка (если соответствующий конечный автомат детерминированный) однозначно выбирает подстановку, т.е. правую часть правила $A \to aB$ либо $A \to b$, заменяющую A в сентенциальной форме.
- Однозначность выбора достигается тем, что в детерминированном автомате в множестве альтернатив каждого нетерминала $A \to a_1B_1 \mid a_2B_2 \mid \ldots \mid a_nB_n$ все терминальные символы a_i различны
- Назовем первый терминал правой части правила ключом этого правила. Для любого автоматного языка может быть построена такая порождающая его автоматная грамматика, что все ключи в различных альтернативах каждого нетерминала различны
- Для любого неавтоматного языка если альтернативы одного и того же нетерминала начинаются РАЗЛИЧНЫМИ терминалами, то выбор правил подстановки тоже однозначен. Такие грамматики без ε-правил называются s-грамматиками

S-грамматики как обобщение автоматных грамматик

- s-грамматики (здесь 's' от слова *simple*, простой) являются простым обобщением автоматных грамматик. s-грамматики имеют вид правил $A \rightarrow a\beta$, где a – терминал (ключ правила), а β - произвольня цепочка из терминалов и нетерминалов, причем у всех альтернатив одного и того же нетерминала ключи различны.
- Это позволяет построить простой детерминированный нисходящий алгоритм распознавания для s-грамматик

Алгоритм нисходящего синтаксического анализа для s-грамматик линейный

- Алгоритм нисходящего синтаксического анализа для s-грамматик реализуется простым алгоритмом со стеком - МП-автоматом с одним состоянием. Алфавит магазина этого МП-автомата включает все терминальные и нетерминальные символы грамматики
  - Вначале в магазин помещается начальный символ грамматики.
  - Очередной терминал входной цепочки и верхний символ магазина определяют, какая цепочка должна быть помещена в магазин
    - Если верхний символ магазина – нетерминал, то очередной входной терминал – это ключ, определяющий правую часть правила грамматики, которой должен быть заменен нетерминал в восстанавливаемом выводе
    - Если верхний символ магазина – терминал, он должен совпадать с очередным терминалом входной цепочки. При таком совпадении терминал выбрасывается из магазина и выполняется сдвиг по входной цепочке – анализатор переходит к следующему терминалу входной цепочки
  - Цепочка распознана, если по исчерпании цепочки магазин пуст
  - Несовпадение терминала в верхушке магазина с очередным терминалом входной цепочки свидетельствует об ошибке во входной строке
  - Ошибка также обнаруживается, если для нетерминала, находящегося в верхушке магазина, очередной входной терминал не совпадает ни с одним ключом правил грамматики для этого нетерминала

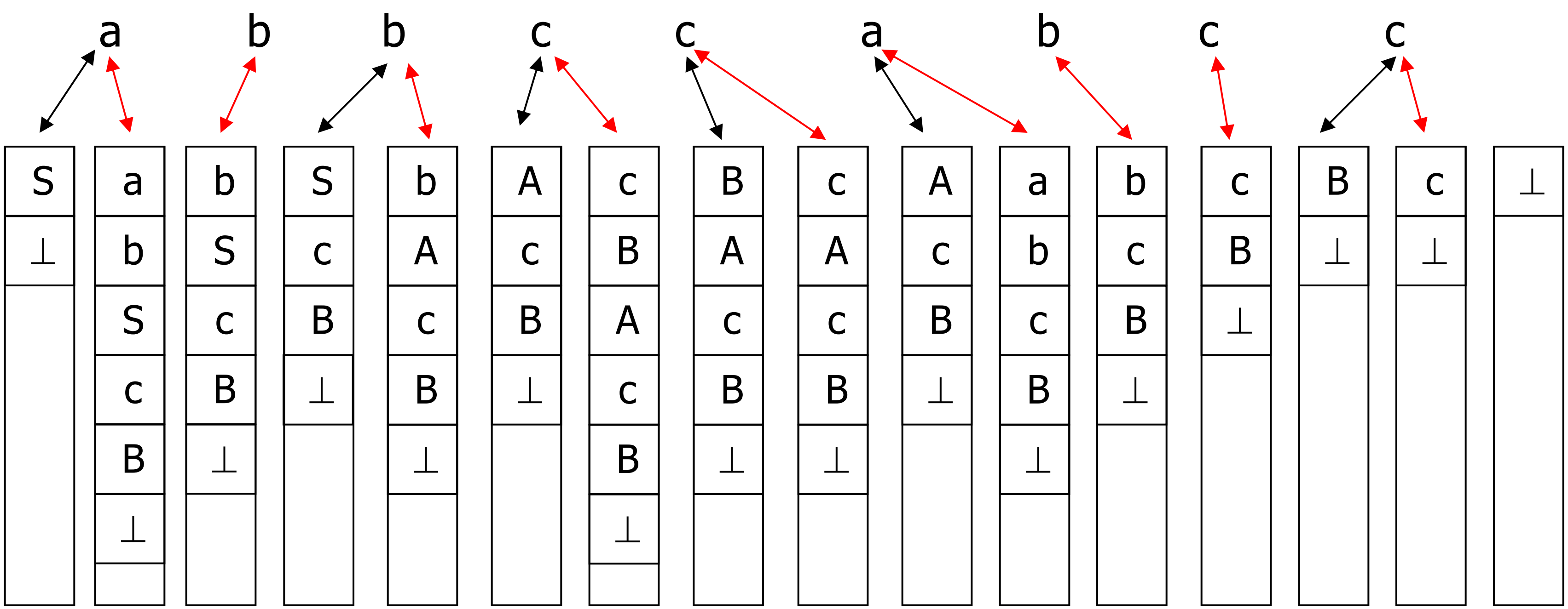
Пример: синтаксический анализ цепочки, порождаемой s-грамматикой

1. S → abScB
2. S → bA
3. A → ab
4. A → cBA
5. B → bBc
6. B → c

S ⇒abScB ⇒abbAcB ⇒ abbcBAcB ⇒
abbccAcB ⇒ abbccabcB ⇒ abbccabcc

	a	b	c
S	abScB	bA	ош
A	ab	ош	cBA
B	ош	bBc	c
a	X	ош	ош
b	ош	X	ош
c	ош	ош	X

В решающей таблице могут быть номера правил, а не правые части правил



Вопрос: алгоритм распознавания проработал – где дерево вывода???

Семантические вычисления для s-грамматик: операционная семантика

Семантики могут быть включены в любом месте правил

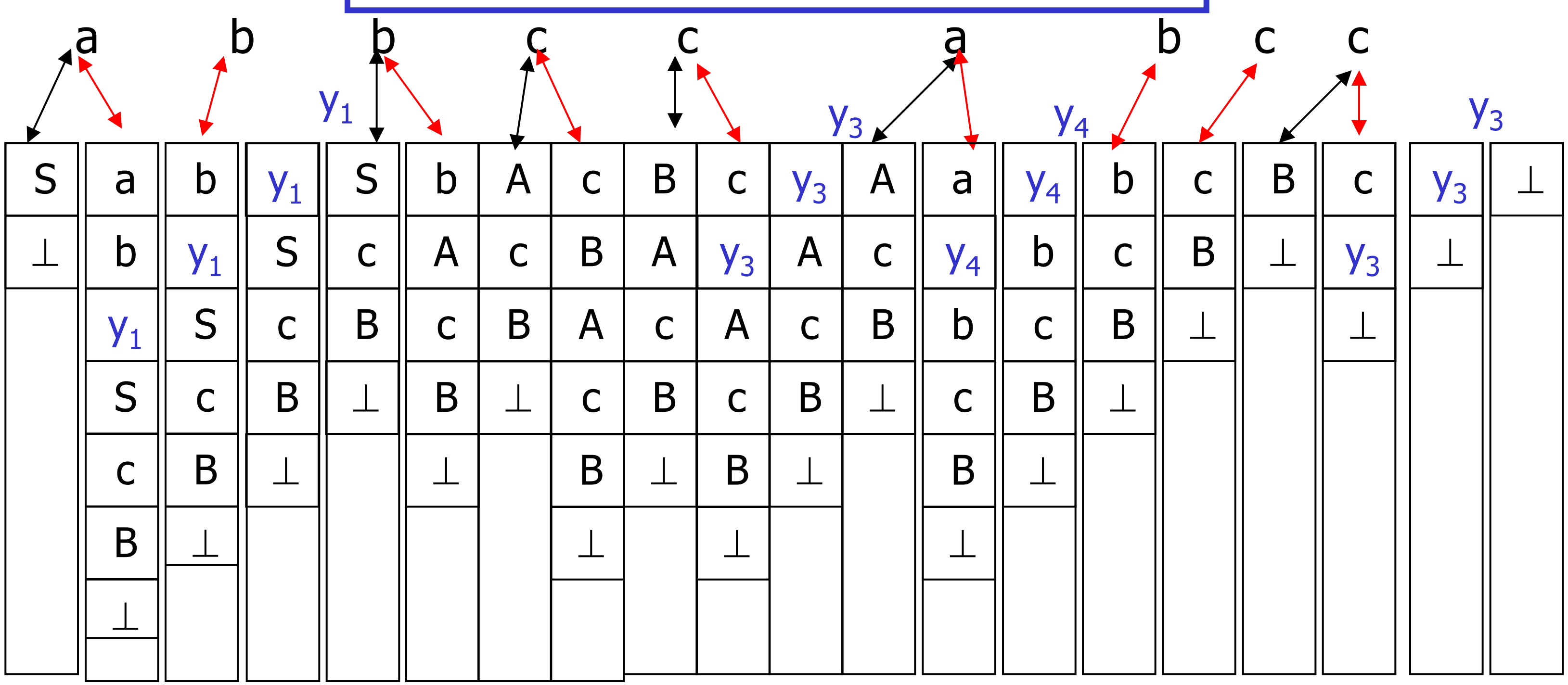
1. $S \to abScB$
2. $S \to bA$
3. $A \to ab$
4. $A \to cBA$
5. $B \to bBc$
6. $B \to c$

$S \Rightarrow abScB \Rightarrow abbAcB \Rightarrow abbcBAcB \Rightarrow abbccAcB \Rightarrow abbccabcB \Rightarrow abbccabcc$

Выполненные семантики для цепочки abbccabcc:

$y_1\, y_3\, y_4\, y_3$

1. $S \to ab\ y_1\ ScB$
2. $S \to b\ A$
3. $A \to a\ y_4\ b$
4. $A \to c\ B\ A$
5. $B \to bB\ y_5\ c$
6. $B \to c\ y_3$



Представления структуры дерева вывода скобочной формой

Грамматика

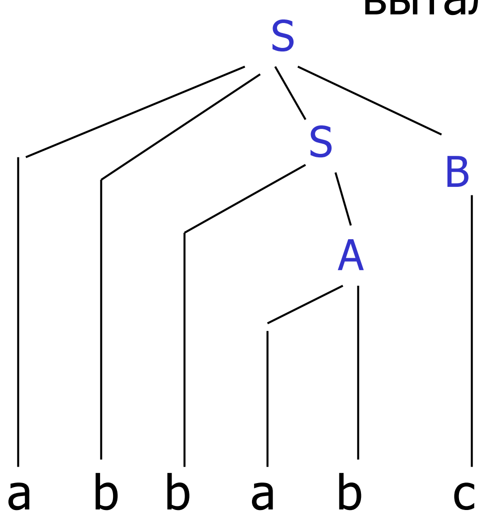
1. $S \to abSB$
2. $S \to bA$
3. $A \to ab$
4. $A \to cBA$
5. $B \to bB$
6. $B \to c$

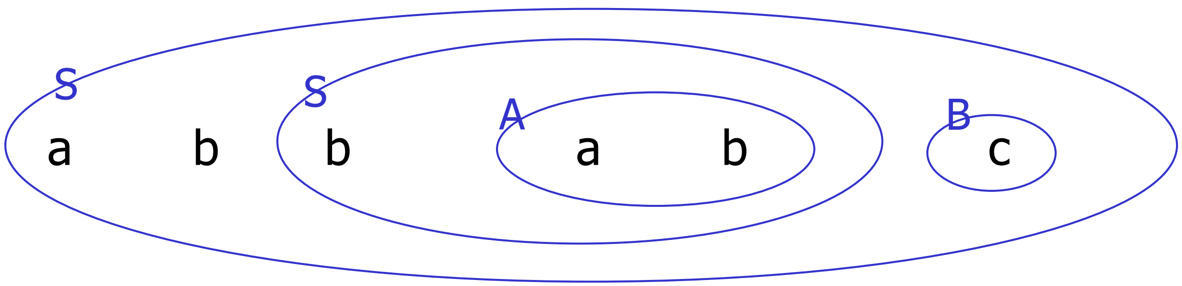
$S \Rightarrow abSB \Rightarrow abbAB \Rightarrow abbabB \Rightarrow abbabc$

Грамматика с семантическими действиями

1. $S \to$ ‘$(_S$‘ abScB ‘$_S)$’
2. $S \to$ ‘$(_S$’ b A ‘$_S)$’
3. $A \to$ ‘$(_A$’ a b ‘$_A)$’
4. $A \to$ ‘$(_A$’ cBA ‘$_A)$’
5. $B \to$ ‘$(_B$’ bB ‘$_B)$’
6. $B \to$ ‘$(_B$’ c ‘$_B)$’

При анализе выводить все выталкиваемые из стека символы

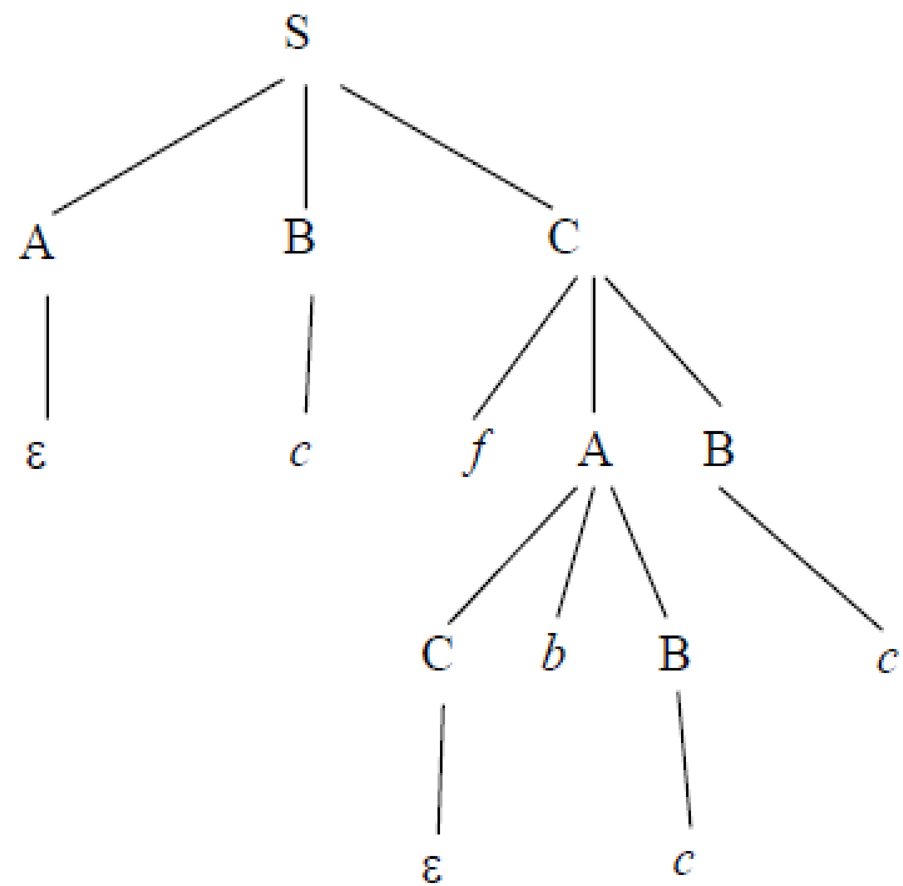




S (a b S (b A (a b)) B (c))

Представление цепочки в виде дерева

- S(A ( ) B(c) C(f A(C ( ) b B(c)) B(c)))

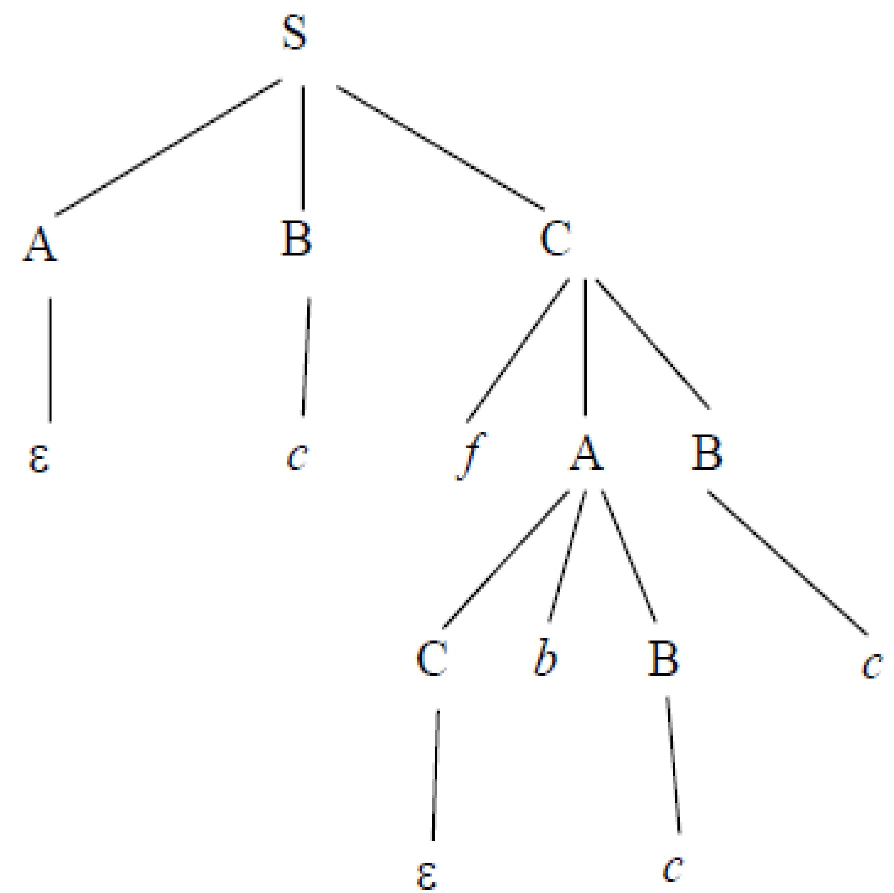


- Полученная на выходе при трансляции последовательность символов представляет структуру входной цепочки – дерево синтаксического анализа в скобочной форме
- При необходимости в качестве семантических действий могут быть выбраны такие, которые представляют дерево в списочной форме, в форме рисунка и т.п.
- Таким образом, само дерево синтаксического разбора ВИРТУАЛЬНО всегда присутствует как результат последовательности применения правил при синтаксическом анализе входной цепочки

Семантические вычисления при нисходящем разборе

- S(A ( ) B(c) C(f A(C ( ) b B(c)) B(c)))

- Трансляция при нисходящем синтаксическом анализе проводится семантическими процедурами, которые обычно вставляются непосредственно в правые части правил грамматики
- При синтаксическом анализе правые части грамматики заталкиваются в магазин вместе с семантическими процедурами на основании таблицы решений. Семантические процедуры выполняются при их выталкивании из магазина
  - Фактически, входная цепочка просто управляет последовательностью выполнения этих семантических процедур при синтаксическом анализаторе, и для каждой входной цепочки такая последовательность будет своей



Обобщение s-грамматик (для k=1)

Пример
1. $S \to bSbAc$
2. $S \to AbBc$
3. $A \to aSc$
4. $A \to cABc$
5. $B \to a$
6. $B \to dD$
7. $D \to Ba$

Правая часть правила 7 начинается не с терминала. Но других альтернатив для D нет!

a lookup table

	a	b	c	d
S	2	1	2	ош
D	7	ош	ош	7

Правая часть правила 2 начинается не с терминала
Но все цепочки, которые можно вывести из цепочки AbBc начинаются с терминалов {a, c}. Следовательно, если очередной нетерминал S, то можно однозначно сделать выбор правила, по которому нужно S заменить, анализируя только один очередной терминал на входе. Множество выбора для правила 1 есть {b}, множество выбора для правила 2 есть {a, c}. Они не пересекаются

Пример
0. $S' \to S\$$
1. $S \to bSaAc$
2. $S \to \varepsilon$
3. $A \to bSc$
4. $A \to cABc$
5. $B \to aBc$
6. $B \to bc$

Правая часть правила 2 – пустая цепочка. Можно ли однозначно определить, следует ли S заменять пустой цепочкой?
Для этого определим множество выбора правила $N \to \alpha$:

a lookup table

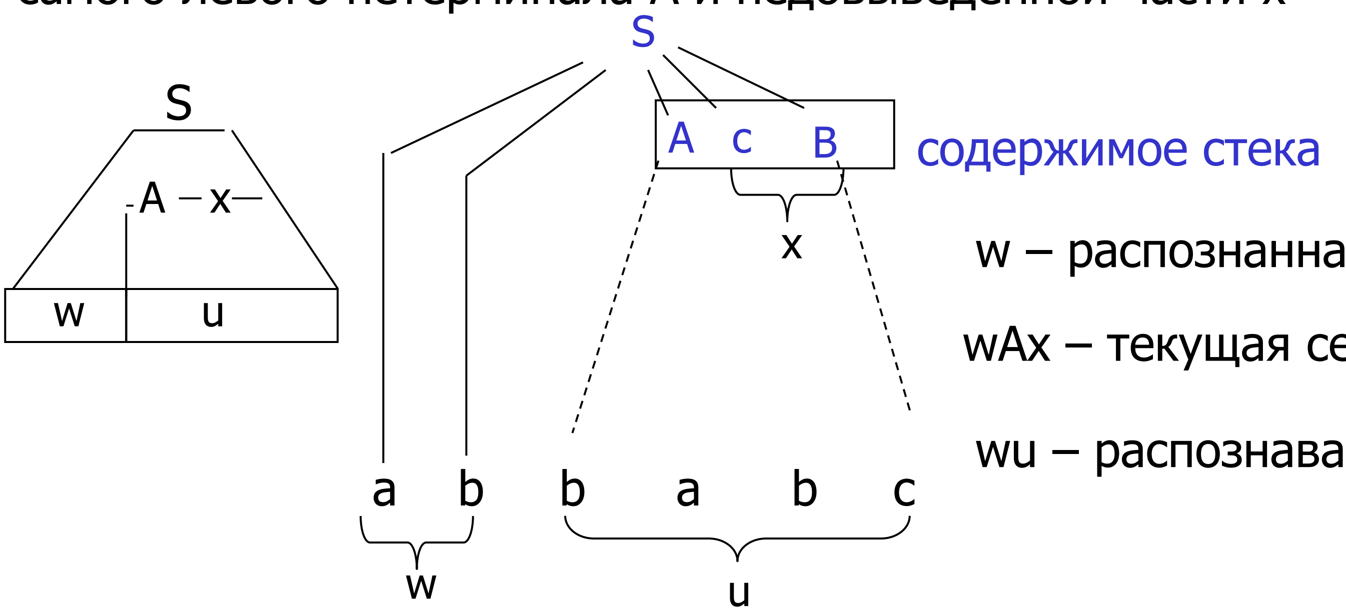
	a	b	c	$
S	2	1	2	2

множество выбора правила $N \to \alpha$ – это терминалы, с которых в сентенциальных формах могут продолжаться цепочки, если нетерминал N заменим на α.

Множество выбора правила $S \to \varepsilon$ - это {$, a, c}. Оно не пересекается с множеством выбора для правила 1, которое есть {b}

LL(k) языки и грамматики

Построение дерева вывода в процессе восстановления левого вывода цепочки. Промежуточная цепочка в процессе вывода состоит из цепочки терминалов wu, самого левого нетерминала A и недовыведенной части x



w – распознанная часть цепочки

wAx – текущая сентенциальная форма

wu – распознаваемая цепочка

Для продолжения разбора требуется заменить нетерминал A по одному из правил вида $A \rightarrow \alpha$. Чтобы разбор был детерминированным (без возвратов), правило требуется выбирать однозначно на основе анализа оставшейся части входной цепочки

Грамматика имеет свойство LL(k), если для однозначного выбора правила достаточно рассмотреть только Ax и первые k символов непросмотренной цепочки u. Обычно используется k=1, т.е. выполнить анализ одного очередного терминального символа оставшейся части входной цепочки u

LL(k) - грамматики

L L (k) - грамматики

просмотр вперед на k символов

восстанавливаем левый вывод

просматриваем цепочку слева

- первая буква L в названии LL(k): входная цепочка читается слева направо
- вторая L: восстанавливается левый вывод входной цепочки
- k означает, что на каждом шаге для принятия решения используется k следующих символов непрочитанной части входной цепочки (look ahead, заглядывание вперед по цепочке)

Теория LL(1)-грамматик

Функция FIRST(α) – множество терминальных символов, с которых могут начинаться цепочки, выводимые из цепочки α (цепоча α может содержать и нетерминальные символы). Если ε выводится из α, то ε включается в FIRST(α).

Если FIRST(α) $\cap$ FIRST(β) = $\varnothing$, то выбор между правилами $A \to \alpha$ и $A \to \beta$ однозначен

Функция FOLLOW(A) – множество терминальных символов, с которых могут начинаться цепочки, которые следуют за A в любых сентенциальных формах. Если A является последним символом сентенциальной формы, то в FOLLOW(A) включается $.

Если FIRST(α) $\cap$ FOLLOW(A) = $\varnothing$, то выбор между правилами $A \to \alpha$ и $A \to \varepsilon$ однозначен

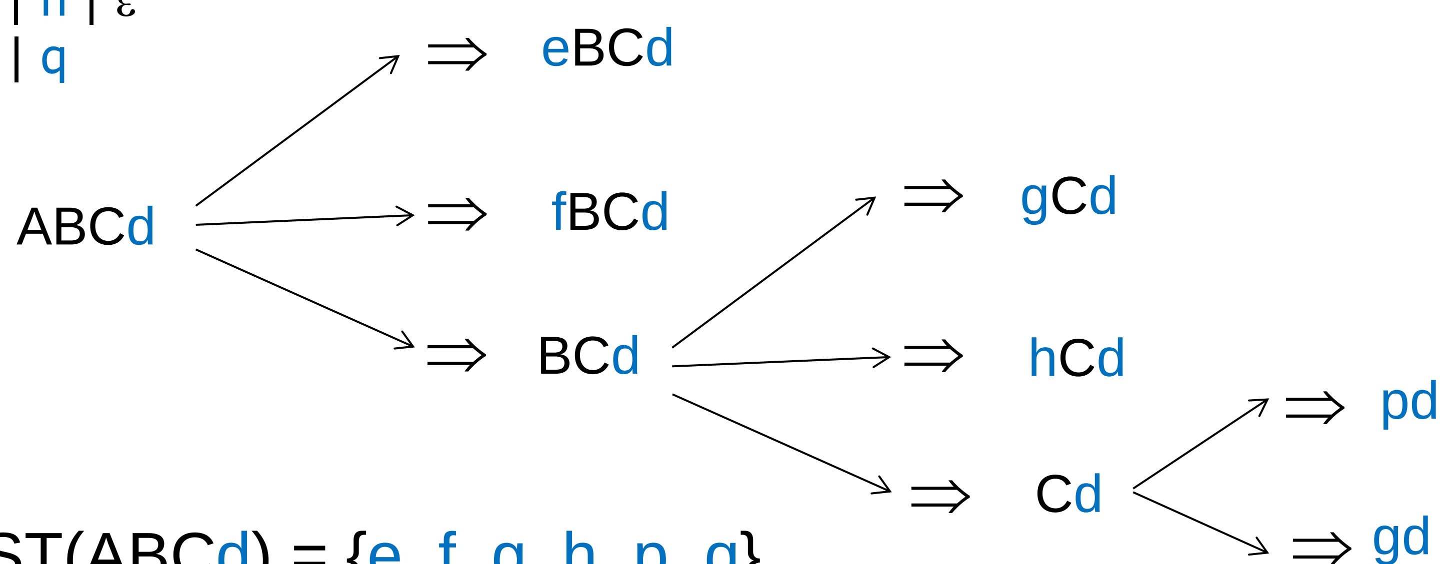
- Грамматика имеет свойство LL(k), если из существования двух цепочек левых выводов:
  - $S \Rightarrow^* wAx \Rightarrow w\alpha x \Rightarrow^* wu$ (если A заменяется по правилу $A \to \alpha$)
  - $S \Rightarrow^* wAx \Rightarrow w\beta x \Rightarrow^* wv$ (если A заменяется по правилу $A \to \beta$)
  - из условия $FIRST_k(u)=FIRST_k(v)$ следует $\alpha = \beta$.
  - $FIRST_k(u)$ – терминальные цепочки длиной k, обобщение функции FIRST(u)
- В случае k=1, для выбора альтернативы для A достаточно знать только нетерминал A и терминал a - очередной символ входной цепочки (первый символ цепочки u):
  - следует выбрать правило $A \Rightarrow \alpha$, если a входит в $FIRST_1(\alpha)$
  - следует выбрать правило $A \Rightarrow \varepsilon$, если a входит в $FOLLOW_1(A)$

Функция FIRST(α)

- FIRST(α)
  - α - произвольная строка из $(T \cup N)^*$ (т.е. строка из терминальных и нетерминальных символов)
  - Результат – множество терминалов
  - Если $\alpha \Rightarrow^* a\beta$, где a – терминал, то $a \in$ FIRST(α)

$S \rightarrow ABCd$
$A \rightarrow e \mid f \mid \varepsilon$
$B \rightarrow g \mid h \mid \varepsilon$
$C \rightarrow p \mid q$

FIRST(ABCd)



FIRST(ABCd) = {e, f, g, h, p, q}

Множество LOOKAHEAD

LL(1)-анализаторы просматривают поток только на один символ вперед при принятии решения о том, какое правило грамматики необходимо применить

Иногда строят LOOKAHEAD таблицу – множество выбора конкретной альтернативы для нетерминала (правила) по очередному терминальному символу

LOOKAHEAD ($A \to \alpha$) = FIRST(α), если неверно, что $\alpha \Rightarrow^* \varepsilon$

LOOKAHEAD ($A \to \alpha$) = FIRST(α) $\cup$ FOLLOW(A), если $\alpha \Rightarrow^* \varepsilon$

Пример:

0. $S' \to S\$$
1. $S \to AbB$
2. $S \to d$
3. $A \to CAb$
4. $A \to B$
5. $B \to cSd$
6. $B \to \varepsilon$
7. $C \to a$
8. $C \to ed$

Вычисление множеств выбора для k=1

1. LOOKAHEAD ($S \to AbB$) = FIRST(AbB) = {a, b, c, e}
2. LOOKAHEAD ($S \to d$) = FIRST(d) = {d}
3. LOOKAHEAD ($A \to CAb$) = FIRST(CAb) = {a, e}
4. LOOKAHEAD ($A \to B$) = FIRST(B) $\cup$ FOLLOW(A) = {c, b}
5. LOOKAHEAD ($B \to cSd$) = FIRST(cSd) = {c}
6. LOOKAHEAD ($B \to \varepsilon$) = FOLLOW(B) = {b, d, \$}
7. LOOKAHEAD ($C \to a$) = FIRST(a) = {a}
8. LOOKAHEAD ($C \to ed$) = FIRST(ed) = {e}

Lookahead table

- LOOKAHEAD $(A\to X_1X_2 \dots X_n) = \cup \{ FIRST(X_i) \mid EMPTY(X_1X_2 \dots X_{i-1}) \}$
  $\cup$ if $EMPTY(X_1X_2 \dots X_n)$ then $FOLLOW(A)$ else $\varnothing$

Пример.
G:: $S' \to S\$$
$S \to BA \mid AAd$
$A \to a \mid \varepsilon$
$B \to bA \mid cB$

N	FIRST(N)	FOLLOW(N)
S'	{ a, b, c, d }	{ }
S	{ a, b, c, d }	{ $ }
A	{ a }	{ a, d, $ }
B	{ b, c }	{ a, $ }

Правила	LOOKAHEAD
$S' \to S\$$	{a, b, c, d, $ }
$S \to BA$	{ b, c }
$S \to AAd$	{ a, d }
$A \to a$	{ a}
$A \to \varepsilon$	{ a, d, $ }
$B \to bA$	{ b }
$B \to cB$	{ c }

Теорема. КС-грамматика G является LL(1)-грамматикой, если и только если для любых двух альтернатив $A\to\alpha$ и $A\to\beta$ любого нетерминала A грамматики G: LOOKAHEAD $(A\to\alpha) \cap$ LOOKAHEAD $(A\to\beta) = \varnothing$

Примеры построения таблицы решений для LL(1)

1. $S' \to S\$$
2. $S \to aSb$
3. $S \to \varepsilon$

FOLLOW(S) = {$, b}

Грамматика LL(1)

Таблица решений

	a	b	$
S′	1	ош	1
S	2	3	3

---

1. $S' \to S\$$
2. $S \to F$
3. $S \to (S + F)$
4. $F \to i$

FIRST (F) = { i }
FIRST ((S+F)) = { (}

Грамматика LL(1)

Пустые клетки – это ошибки, их обычно не пишут в таблице

	(	)	+	i	$
S′	1			1	
S	3			2	
F				4	

Грамматика, порождающая подмножество типов языка Pascal:

1. <type> → <simple>
2. <type> → ^id
3. <type> → array [<simple>] of <type>
4. <simple> → integer
5. <simple> → char
6. <simple> → num .. num

	^	array	id	ch	num
<type>	2	3	1	1	1
<simple>	ош	ош	4	5	6

Здесь ошибки указаны явно

JFLAP: http://www.jflap.org/ учебник по LL(1) синтаксическому анализу

Пример: прямая левая рекурсия

1. $E \to E+T$
2. $E \to T$
3. $T \to T*P$
4. $T \to P$
5. $P \to i$
6. $P \to (E)$

$A \to \alpha$	FIRST(α)
1. $E \to E+T$	{ i, (}
2. $E \to T$	{ i, (}
3. $T \to T*P$	{ i, (}
4. $T \to P$	{ i, (}
5. $P \to i$	{ i }
6. $P \to (E)$	{ (}

Грамматика не является LL(1), т.к. правила для E и T содержат прямую левую рекурсию

$E \to E+T \mid T$

заменяем:

$E \to TE_1 \quad E_1 \to +TE_1 \mid \varepsilon$

или в БНФ:

E ::= T {+T}

0. $S \to E\ \$$
1. $E \to TE_1$
2. $E_1 \to +TE_1$

2a. $E_1 \to \varepsilon$

3. $T \to PT_1$
4. $T_1 \to *T_1$

4a. $T_1 \to \varepsilon$

5. $P \to i$
6. $P \to (E)$

Правило	LOOKAHEAD
1. $E \to TE_1$	{ i, (}
2. $E_1 \to +TE_1$	{ + }
2a. $E_1 \to \varepsilon$	{), \$ }
4. $T_1 \to *T_1$	{ * }
4a. $T_1 \to \varepsilon$	{+,), \$}
5. $P \to i$	{ i }
6. $P \to (E)$	{ (}

таблица решений

	+	*	i	(	)	\$
S			0	0		
E			1	1		
E_1	2				2a	2a
T_1	4a	4			4a	4a
P			5	6		

По 2 альтернативы - три символа, P, E_1 и T_1

Пустые клетки – это ошибки

Пример: трансляция оператора присваивания

0. S $\rightarrow$ i:=E \$ $y_0(i)$
1. E $\rightarrow$ TE_1
2. $E_1 \rightarrow +TE_1$ y_2
2a. $E_1 \rightarrow \varepsilon$
3. T $\rightarrow$ PT_1
4. $T_1 \rightarrow *T_1$ y_4
4a. $T_1 \rightarrow \varepsilon$
5. P $\rightarrow$ i $y_5(i)$
6. P $\rightarrow$ (E)

	+	*	i	(	)	\$
E_1	2				2a	2a
T			3	3		
T_1	4a	4			4a	4a
P			5	6		

Пустые клетки – это ошибки

Трансляция
a:=b+c*d \$
должна дать:

LOAD addr b
LOAD addr c
LOAD addr d
MUL
ADD
STORE addr a

Выберем
семантики:

$y_0(i)$: Gen STORE addr i
y_2: Gen ADD
y_4: Gen MUL
$y_5(i)$: Gen LOAD addr i

a := b + c * d \$

$y_5(b)$

S	a	:=	E	T	P	b	$y_5(b)$	T_1	E_1	+	T	P	c
⊥	:=	E	\$	E_1	T_1	$y_5(b)$	T_1	E_1	$y_0(a)$	T	E_1	T_1	$y_5(c)$
	E	$y_0(a)$	$y_0(a)$	$y_0(a)$	E_1	T_1	E_1	$y_0(a)$	\$	E_1	y_2	E_1	T_1
	$y_0(a)$	\$	⊥	\$	$y_0(a)$	E_1	$y_0(a)$	\$	⊥	y_2	$y_0(a)$	y_2	E_1
	\$	⊥		⊥	\$	$y_0(a)$	\$	⊥		$y_0(a)$	\$	$y_0(a)$	y_2
	⊥				⊥	\$	⊥			\$	⊥	\$	$y_0(a)$

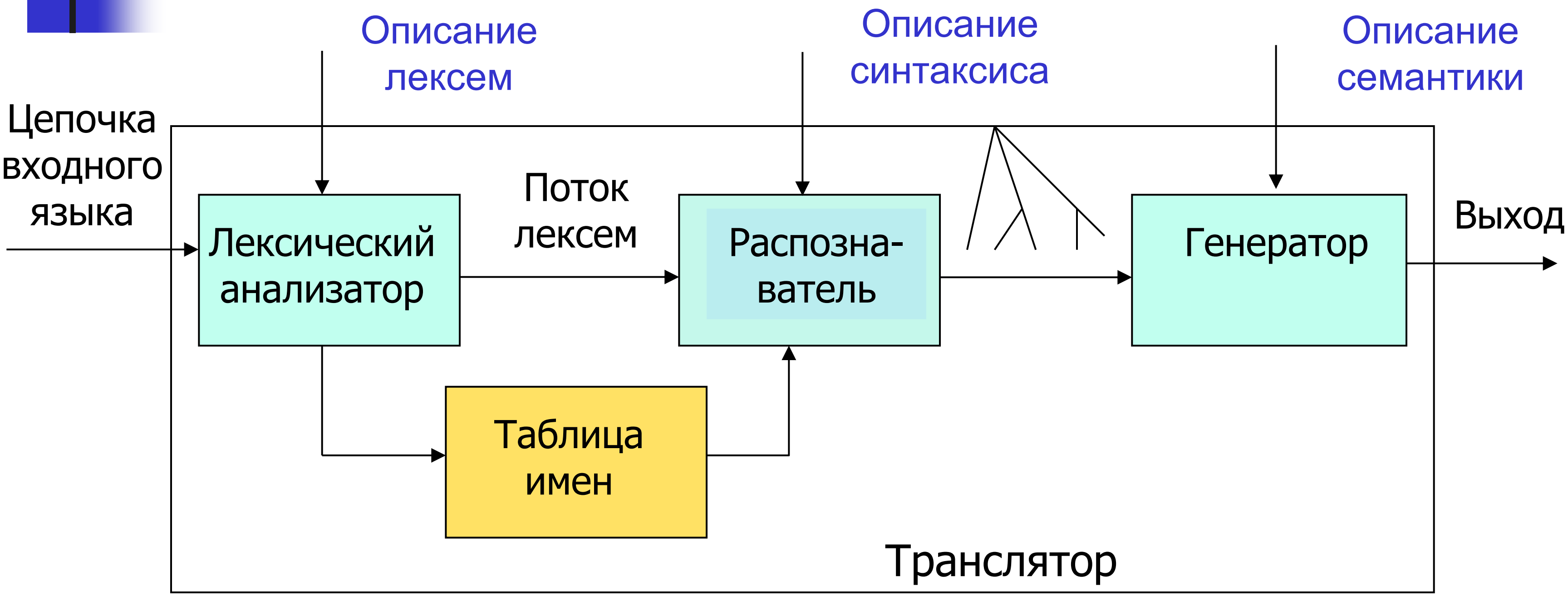
Семантические вычисления

- Семантические вычисления при нисходящем синтаксическом анализе (LL-анализатор) выполняются просто
- Наследуемые атрибуты символов правой части правила сразу вычисляются при замене нетерминала по решающей таблице

Действия СА при обнаружении ошибки

- Что, если в таблице решений синтаксический анализатор попадает на "ошибку"? Это означает, что входная строка НЕ выводится в данной грамматике, иными словами во входной строке синтаксическая ошибка
- При обнаружении ошибки синтаксический анализатор может просто остановиться и указать место возникновения ошибки. Эта стратегия неудобна, поскольку при длинной программе количество повторных запусков на трансляцию может быть большим
- Другая стратегия:
  1) определяем множество "маяков" (синхронизирующих лексем), которые ограничивают операторы, блоки, процедуры и т.п.
  2) при обнаружении синтаксической ошибки СА:
     а) пропускает лексемы входной строки по одной, пока не будет найден маяк, и
     б) управление в анализаторе передается в ближайшую точку анализатора, после которой встречается эта лексема
- Обычно в множество синхронизирующих лексем включают разделители, например, ;,), end или }.. . Набор таких лексем определяется разработчиком анализируемого языка, для которого строится компилятор
- При такой стратегии восстановления может оказаться, что много лексем будет пропущено без проверки на наличие дополнительных ошибок
- Такая стратегия восстановления наиболее проста в реализации
- Существуют и другие стратегии восстановления СА после обнаружения ошибки во входном тексте
- Задача для курсовой работы: почитать о стратегиях восстановления анализа после обнаружения синтаксической ошибки и реализовать ее в компиляторе для Милана

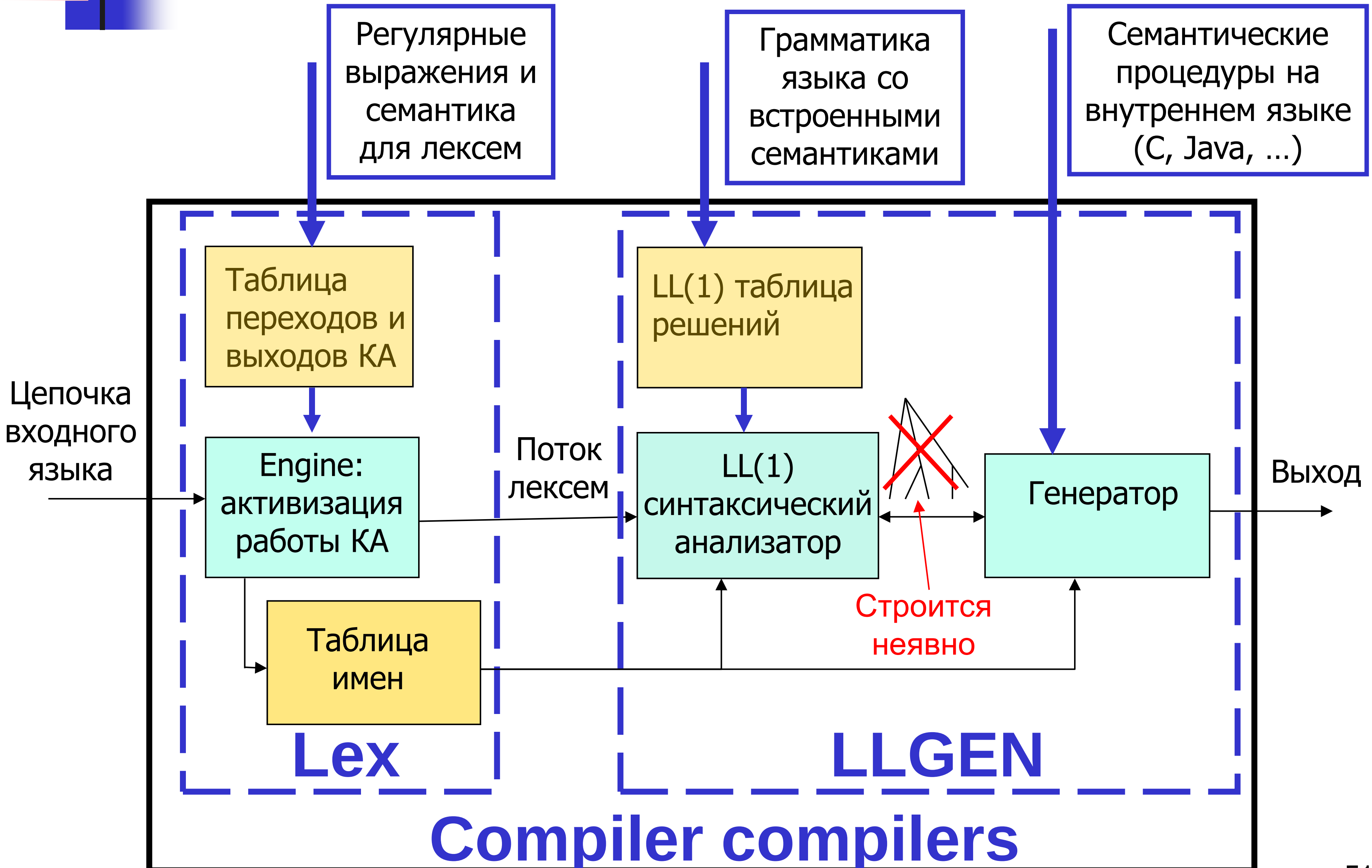
Генератор компиляторов



Можно ли построить такую программную систему для любой грамматики?

- Да, если:
  - задать описание лексических единиц (лексем)
  - задать описание КС-грамматики исходного языка (ограничиться классом КС-грамматик, для которого есть алгоритм СА)
  - задать описание семантики, связанной с правилами грамматики

Компилятор компиляторов для LL-грамматик



Генераторы лексических и синтаксических анализаторов

http://www.kulichki.com/kit/tools/lexparse.html

- **LEX -** классический генератор лексических анализаторов AT&T, поставляемый с Unix
- **YACC (Yet Another Compiler Compiler) -** классический генератор синтаксических анализаторов AT&T, поставляемый с Unix
- **FLEX -** GNU версия генератора сканеров Lex
- **BISON -** GNU версия Yacc
- **ACCENT** компилятор компиляторов, не накладывающий никаких ограничений на грамматики: никакой адаптации к специфическим методам синтаксического анализа, таким как LL(k) или LALR(k), не требуется. Поддерживает расширенную БНФ
- **AFLEX & AYACC -** аналогичны юниксовым инструментам Lex и Yacc, но написаны на Ада и генерируют на выходе компилятор на языке Ада
- **ANAGRAM -** генератор синтаксических анализаторов LALR с возможностью автоматического восстановления после синтаксических ошибок
- **BTYACC** - модифицированная версия Yacc, поддерживающая автоматический откат и семантическое устранение неоднозначности для разбора неоднозначных грамматик
- **BYACC (**Berkeley Yacc) - генератор синтаксических анализаторов LALR(1)
- **COCO/R** генерирует анализаторы, работающие по методу рекурсивного спуска и связанные с ними сканеры по атрибутным грамматикам
- **SCANGEN, LLGEN, LALRGEN -** генераторы лексических, LL(1) и LALR(1) генераторов
- **TP LEX AND YACC -** генератор лексических и синтаксических анализаторов для Turbo Pascal

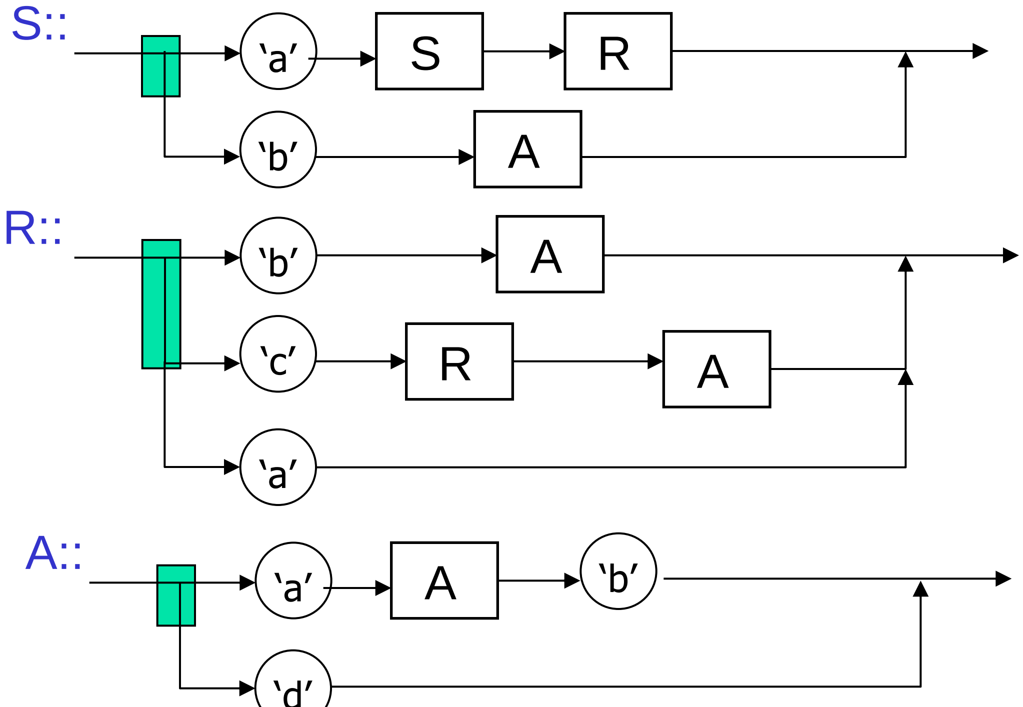
... ... Подобных генераторов десятки (если не сотни ...)

Грамматики рекурсивного спуска

Определение

- Анализатором рекурсивного спуска называется вариант предсказывающего LL(1) анализа, в котором каждому нетерминалу сопоставляется процедура (вообще говоря, рекурсивная), и стек образуется неявно при вызовах таких процедур
- Грамматикой рекурсивного спуска называется LL(1)-грамматика, которая представляется синтаксическими диаграммами для каждого нетерминала такими, что в любом разветвлении каждой синтаксической диаграммы выбор альтернативного пути может быть сделан на основе анализа одного очередного терминального символа входной цепочки
- Очевидно, что любая s-грамматика является грамматикой рекурсивного спуска

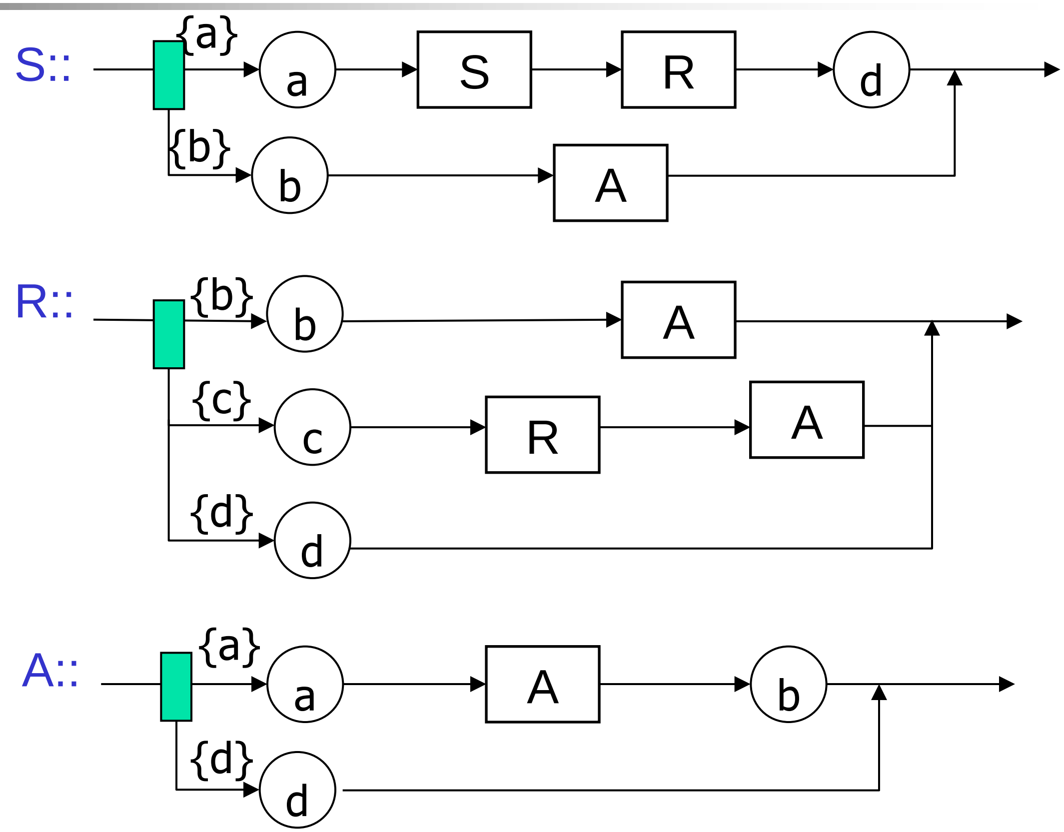
1. $S \to aSR$
2. $S \to bA$
3. $R \to bA$
4. $R \to cRA$
5. $R \to a$
6. $A \to aAb$
7. $A \to d$



Пример: s-грамматика и грамматика рекурсивного спуска

1. S → aSRd
2. S → bA
3. R → bA
4. R → cRA
5. R → d
6. A → aAb
7. A → d

В фигурных скобках - правило выбора пути по очередному терминалу



- Для каждого нетерминала строится своя распознающая процедура
- Альтернативные пути в синтаксической диаграмме помечены различными терминалами, следовательно при анализе цепочки путь вычислений в процедурах будет выбран однозначно по результату анализа одного очередного терминального символа
- При синтаксическом анализе для таких грамматик стек не строится. Стек образуется неявно при рекурсивных вызовах процедур в процессе анализа входной цепочки

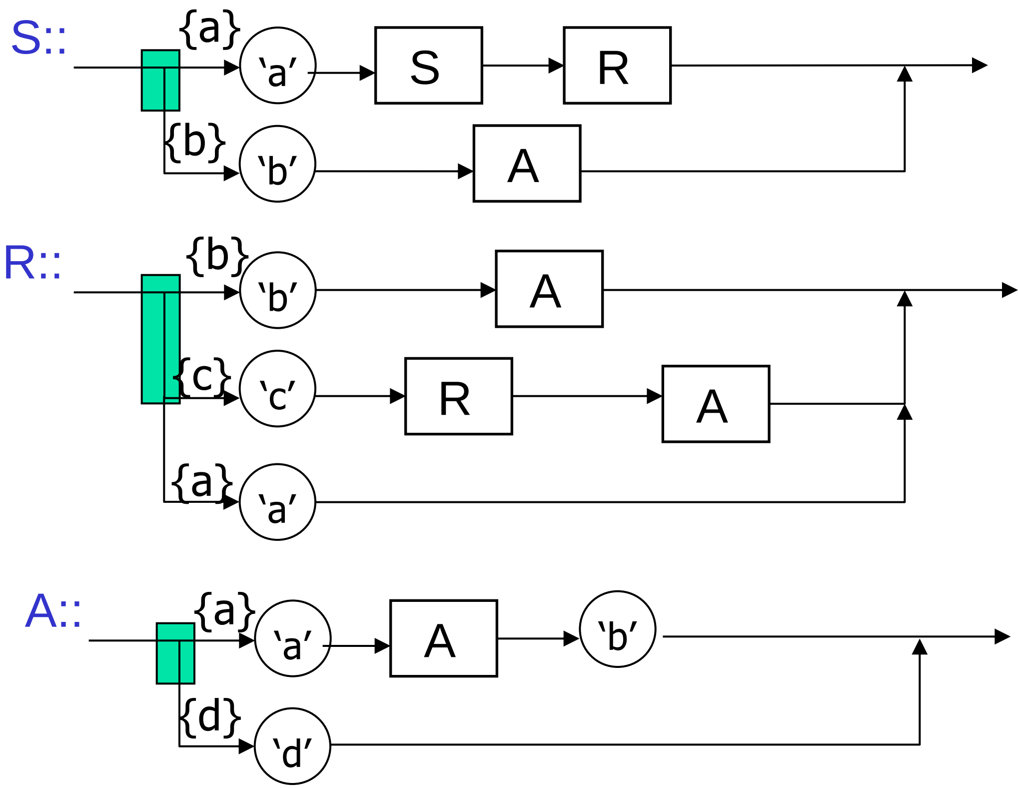
Идея алгоритма рекурсивного спуска

```
int Ук:=0; // указ-ль на очередной символ

S ( ) {
  if( s[Ук] == 'a') then { Ук++; S(); R(); }
  else
      { if( s[Ук] == 'b') then { Ук++; A() }
            else { Error1( ) ; }
      }
}

A ( ) {
 if( s[Ук] == 'a') then {
    Ук++;  A( );
    if( s[Ук] == 'b') then { Ук++; break; }
          else { Error2( ); }
      }
 else { if( s[Ук] == 'd') then {Ук++; break; }
          else { Error3( ) };
       }
}
```

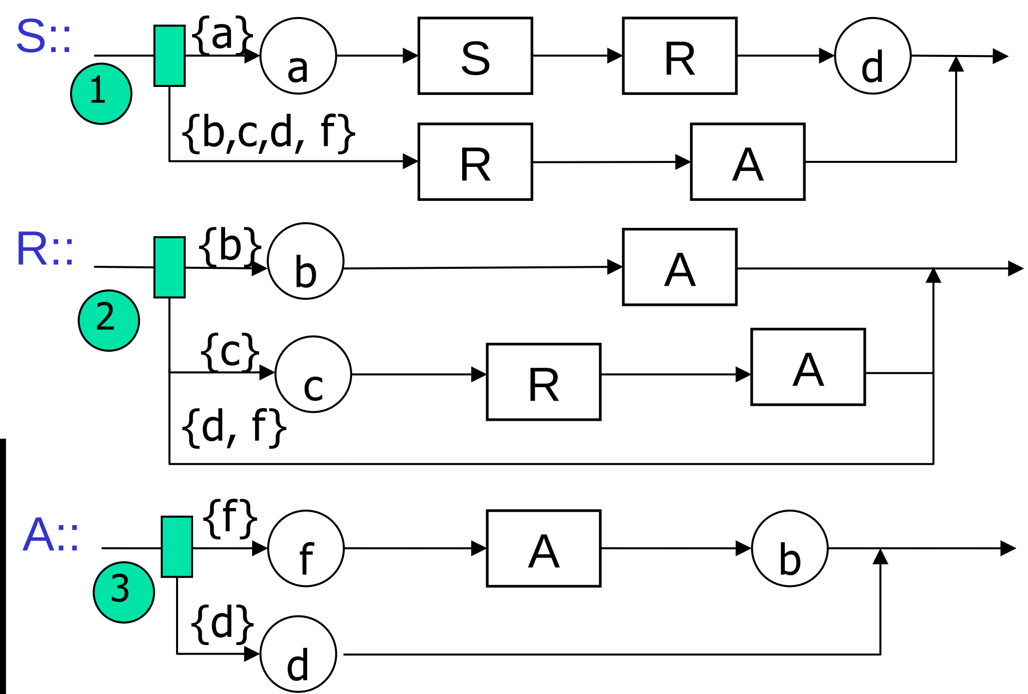
При входе в процедуру N Ук установлен на первый символ подцепочки, выводящейся из нетерминала N



- Каждому нетерминалу сопоставляется процедура. Начальному нетерминалу – main
- Каждая процедура распознает максимальную подстроку своего языка

Пример грамматики рекурсивного спуска (не s-грамматики)

1. $S \to aSRd$
2. $S \to RA$
3. $R \to bA$
4. $R \to cRA$
5. $R \to \varepsilon$
6. $A \to fAb$
7. $A \to d$



В методе рекурсивного спуска поскольку одна из альтернатив R – пустой символ, нужно проверить, какой символ идет после нетерминала R (Follow(R) и проверить, пересекаются ли множества

- Выбор 3: определяется очередным терминалом f или d. Выбор однозначен
- Выбор 2: третий путь определяется терминалами Follow(R), т.е. теми терминалами, которые могут встретиться **после** R. Это множество {f, d} тех, с которых начинаются цепочки, выводимые из A, поскольку после R стоит только A. Выбор пути однозначен
- Выбор 1: второй путь определяется терминалами {b, c, f, d}, т.е. теми, с которых могут начаться цепочки, выводимые из R, и теми, которые могут встретиться после R, поскольку есть правило $R \to \varepsilon$. Выбор однозначен

Pages: 23, 13-54, 128-97

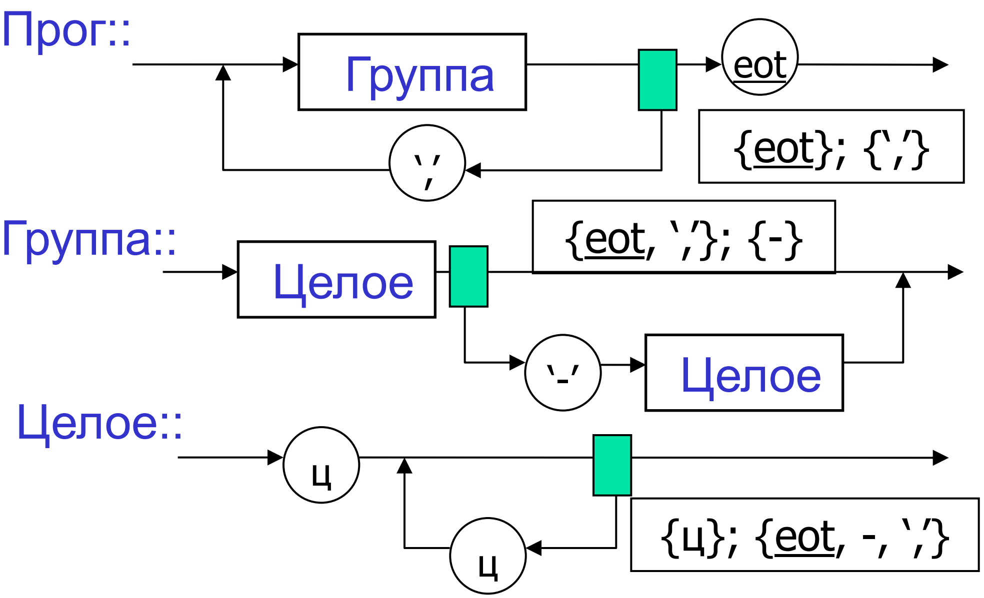
Распознаватель языка перечисления номеров страниц

```
int Ук:=0; // указ-ль на очередной символ

main( ) {
m: Группа( );
   if( s[Ук] == 'eot') then { Ук++; break };
     else if( s[Ук] == ',') then { Ук++; goto m };
     else Error1( ) };

Группа( ) {
 Целое( );
 if( s[Ук] == '-') then { Ук++; Целое() };
    else skip
};

Целое( ) {
      if( s[Ук] == 'ц') then Ук++;
      else Error2( );
  m: if( s[Ук] == 'ц') then { Ук++; goto m};
     else skip
};
```



Для каждого нетерминала – своя распознающая процедура

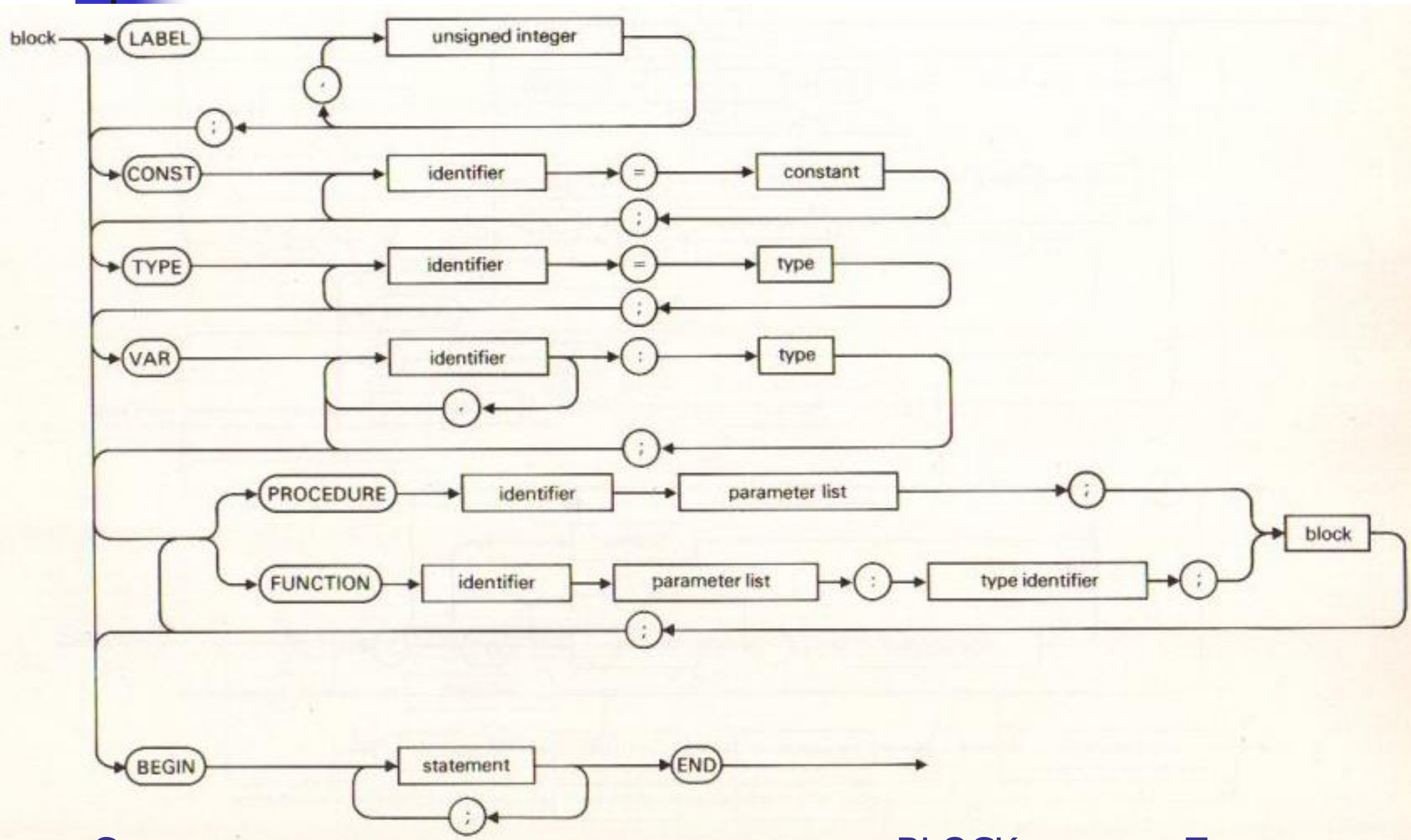
Эта грамматика рекурсивного спуска: в каждом разветвлении синтаксической диаграммы по очередному терминалу путь определяется однозначно

Добавление семантических операций в синтаксическую диаграмму позволяет построить транслятор

Lookahead символы

- Стандартный метод, примененный в трансляции рекурсивного спуска, заключается в использовании одиночного, упреждающего символа, объявленного глобально (Lookahead символа)
- В рамках процедуры инициализации обязательно выполняется функция, считывающая первый символ из входного потока, например, GetChar( ). Каждый вызов функции GetChar( ) или аналогичной дает следующий символ из входного потока

Пример синтаксической диаграммы, в которой выбор пути при распознавании определяется однозначно



- Синтаксическая диаграмма конструкции <BLOCK> языка Паскаль

Семантические вычисления при рекурсивном спуске

- При распознавании цепочки могут выполняться семантические процедуры
- Синтезируемые атрибуты нетерминалов являются определяемыми параметрами при вызове соответствующих процедур.
- Наследуемые атрибуты символов являются параметрами, устанавливаемыми при вызове соответствующей процедуры
- Пример: генерация кода. Пусть нетерминал имеет два атрибута:
  N.Снач – начальный адрес фрагмента программы, который вычисляет N,
  N.Скон – адрес, следующий за фрагментом, вычисляющим N.

  Тогда C можно считать не глобальным, а локальным параметром, в main устанавливается Снач=0, а при вызовах текущее C передается как параметр Снач, а после вызова процедуры очередного нетерминала локальное C устанавливается по значению Скон.
- Пример – язык линейных алгебраических уравнений. Там как параметры передаются наследуемые атрибуты (например, номер обрабатываемого уравнения)

Заключение

- Задачей синтаксического анализа КС-языка является восстановление вывода (дерева вывода) цепочки языка из начального символа грамматики. Теория синтаксического анализа является наиболее разработанной областью информатики
- Существуют универсальные неэффективные алгоритмы синтаксического анализа КС-языков, для подклассов КС-грамматик существуют эффективные (линейные) алгоритмы распознавания
- Прежде чем строить синтаксический анализатор, в грамматике следует выбросить неприменяемые правила
- Существует два класса алгоритмов синтаксического анализа КС-языков: НИСХОДЯЩИЕ и ВОСХОДЯЩИЕ (по синтаксическому дереву) алгоритмы. Нисходящие алгоритмы восстанавливают ЛЕВЫЙ вывод цепочки языка, восходящие – ПРАВЫЙ вывод
- Эффективные нисходящие алгоритмы синтаксического анализа для:
  - s-грамматик,
  - LL(k)-грамматик,
  - грамматик рекурсивного спуска
- Алгоритм рекурсивного спуска часто используется при разработке трансляторов специализированных языков

Спасибо за внимание