Московский государственный
университет им. М.В. Ломоносова
Факультет вычислительной математики и кибернетики

И. А. Волкова, А. А. Вылиток, Т. В. Руденко

Формальные грамматики и языки. Элементы теории трансляции

учебное пособие для студентов II курса

(издание третье, переработанное и дополненное)

Москва
2008

УДК 519.682.1+681.142.2

Приводятся основные определения, понятия и алгоритмы теории формальных грамматик и языков, некоторые методы трансляции, а также наборы задач по каждой из рассматриваемых тем. Излагаемые методы трансляции проиллюстрированы на примере модельного языка.

В третьем издании все примеры реализации методов и алгоритмов приводятся на языке C++, изменен и расширен набор задач по всем разделам.

Для студентов факультета ВМК в поддержку основного лекционного курса «Системы программирования» и для преподавателей, ведущих практические занятия по этому курсу.

Рецензенты:

доц. Корухова Л.С.

Волкова И. А., Вылиток А. А., Руденко Т. В. «Формальные грамматики и языки. Элементы теории трансляции. (учебное пособие для студентов II курса)» — издание третье (переработанное и дополненное)

Издательский отдел факультета ВМиК МГУ

Печатается по решению Редакционно-издательского Совета факультета вычислительной математики и кибернетики МГУ им. М. В. Ломоносова

ISBN 5-89407-032-5

Элементы теории формальных языков и грамматик

Введение

В этом разделе изложены некоторые аспекты теории формальных языков, существенные с точки зрения трансляции. Здесь введены базовые понятия и даны определения, связанные с одним из основных механизмов определения языков — грамматиками, приведена наиболее распространенная классификация грамматик (по Хомскому). Особое внимание уделяется контекстно-свободным и регулярным грамматикам. Грамматики этих классов широко используются при трансляции языков программирования. Здесь не приводятся доказательства корректности алгоритмов и сформулированных фактов, свойств, теорем; их можно найти в книгах, указанных в списке литературы.

Основные понятия и определения

Определение: *алфавит* — это конечное множество символов.

Предполагается, что термин «символ» имеет достаточно ясный интуитивный смысл и не нуждается в дальнейшем уточнении.

Определение: *цепочкой символов в алфавите V* называется любая конечная последовательность символов этого алфавита.

Определение: цепочка, которая не содержит ни одного символа, называется *пустой цепочкой*. Для ее обозначения будем использовать греческую букву «ε».

Предполагается, что сама буква «ε» в алфавит V не входит; она лишь помогает обозначить пустую последовательность.

Определение: если α и β — цепочки, то цепочка $\alpha\beta$ (результат приписывания цепочки β в конец цепочки α), называется *конкатенацией* (или *сцеплением*) цепочек α и β. Конкатенацию можно считать двуместной операцией над цепочками: $\alpha\cdot\beta = \alpha\beta$.

Например, если $\alpha = ab$ и $\beta = cd$, то $\alpha\cdot\beta = abcd$.

Для любой цепочки α справедливы равенства: $\alpha\cdot\varepsilon = \varepsilon\cdot\alpha = \alpha$.

Для любых цепочек α, β, γ справедливо $(\alpha\cdot\beta)\cdot\gamma = \alpha\cdot(\beta\cdot\gamma) = \alpha\beta\gamma$ (свойство ассоциативности операции конкатенации).

Определение: *обращением* (или *реверсом*) цепочки α называется цепочка, символы которой записаны в обратном порядке.

Обращение цепочки α будем обозначать α^R.

Например, если $\alpha = abcdef$, то $\alpha^R = fedcba$.

Для пустой цепочки: $\varepsilon^R = \varepsilon$.

Определение: n-ой степенью цепочки α (будем обозначать α^n) называется конкатенация n цепочек α:

$$\alpha^n = \underbrace{\alpha\alpha \ldots \alpha\alpha\alpha}_{n}$$

$$\alpha^0 = \varepsilon;\ \alpha^n = \alpha\alpha^{n-1} = \alpha^{n-1}\alpha$$

Определение: *длина цепочки* — это число составляющих ее символов (или длина последовательности символов).

Например, если $\alpha = abbcaad$, то длина α равна 7.

Длину цепочки α будем обозначать $|\alpha|$. Длина ε равна 0.

Определение: через $|\alpha|_s$ обозначают *число вхождений* символа *s* в цепочку α.

Например, $|babb|_a = 1$, $|babb|_b = 3$, $|babb|_c = 0$.

Определение: *язык* в алфавите V — это подмножество цепочек конечной длины в этом алфавите.

Определение: обозначим через V^* множество, содержащее все цепочки конечной длины в алфавите V, включая пустую цепочку ε.

Например, если $V = \{0, 1\}$, то $V^* = \{\varepsilon, 0, 1, 00, 11, 01, 10, 000, 001, 011, ...\}$.

Определение: обозначим через V^+ множество, содержащее все цепочки конечной длины в алфавите V, исключая пустую цепочку ε.

Следовательно, $V^* = V^+ \cup \{\varepsilon\}$.

Ясно, что каждый язык L в алфавите V является подмножеством множества V^*, т. е. $L \subseteq V^*$.

Известны различные способы описания языков [3]. Конечный язык можно описать простым перечислением его цепочек. Поскольку формальный язык может быть и бесконечным, требуются механизмы, позволяющие конечным образом представлять бесконечные языки. Можно выделить два основных подхода для такого представления: механизм распознавания и механизм порождения (генерации).

Механизм распознавания (*распознаватель*), по сути, является процедурой (или алгоритмом) специального вида, которая по заданной цепочке определяет, принадлежит ли она языку. Если принадлежит, то процедура останавливается с ответом «да», т. е. *допускает* цепочку, иначе — останавливается с ответом «нет» или зацикливается. Язык, определяемый распознавателем — это множество всех цепочек, которые он допускает.

Примеры распознавателей:

- Машина Тьюринга (МТ). Язык, который можно задать с помощью МТ, называется *рекурсивно перечислимым*. Вместо МТ можно использовать эквивалентные алгоритмические схемы: нормальный алгоритм Маркова (НАМ), машину Поста и др.

- Линейно ограниченный автомат (ЛОА). Представляет собой МТ, в которой лента не бесконечна, а ограничена длиной входного слова[1]. Определяет контекстно-зависимые[2] языки.
- Автомат с магазинной (внешней) памятью (МП-автомат). В отличие от ЛОА, головка не может изменять входное слово и не может сдвигаться влево; имеется дополнительная бесконечная память (*магазин,* или стек), работающая по дисциплине LIFO. Определяет контекстно-свободные языки.
- Конечный автомат (КА). Отличается от МП-автомата отсутствием магазина. Определяет регулярные языки.

Основной способ реализации механизма порождения — использование порождающих грамматик, которые иногда называют грамматиками Хомского. На изучении порождающих грамматик мы и остановимся подробно, и именно этот способ описания языков чаще всего будем использовать в дальнейшем.

Отметим, что не каждый формальный язык можно задать с помощью конечного описания. Действительно, само описание можно рассматривать как цепочку в некотором расширенном алфавите. Следовательно, множество описаний языков счётно, так как множество всех цепочек счётно. Каждый формальный язык в алфавите V является подмножеством счётного множества V^*. Из теории множеств известно, что множество всех подмножеств счётного множества является несчётным. Таким образом, мощность множества формальных языков больше мощности множества конечных описаний и, следовательно, не каждый язык представи́м в виде конечного описания.

Определение: *декартовым произведением* $A \times B$ множеств A и B называется множество $\{ (a, b) \mid a \in A, b \in B \}$.

Определение: *порождающая грамматика* G — это четверка $\langle T, N, P, S \rangle$, где

- T — алфавит терминальных символов (терминалов);
- N — алфавит нетерминальных символов (нетерминалов), $T \cap N = \varnothing$;
- P — конечное подмножество множества $(T \cup N)^+ \times (T \cup N)^*$; элемент (α, β) множества P называется *правилом вывода* и записывается в виде $\alpha \to \beta$; α называется *левой частью* правила, β — *правой частью*; левая часть любого правила из P обязана содержать хотя бы один нетерминал;
- S — начальный символ (цель) грамматики, $S \in N$.

Для записи правил вывода с одинаковыми левыми частями

$$\alpha \to \beta_1 \; \alpha \to \beta_2 \; ... \; \alpha \to \beta_n$$

будем пользоваться сокращенной записью

$$\alpha \to \beta_1 \mid \beta_2 \mid ... \mid \beta_n.$$

---

1) ЛОА является недетерминированной машиной: есть возможность выполнения в какой-то момент двух или более разных команд (то есть выбор команды не детерминирован); с этого момента ЛОА ведет два или более параллельных вычисления. Ответ «да» ЛОА дает, если хотя бы одно из вычислений над входной цепочкой успешно завершается. Известен алгоритм, позволяющий по любой недетерминированной МТ построить эквивалентную детерминированную МТ. Однако до сих пор открыт вопрос, существует ли для любого ЛОА эквивалентный детерминированный ЛОА.

2) Термин «рекурсивно перечислимый» происходит из теории рекурсивных функций, являющейся, также как НАМ и МТ, одной из формализаций понятия «вычислимости» или алгоритма. Смысл остальных названий, характеризующих распознаваемые языки, будет пояснен ниже.

Каждое β_i (i = 1, 2, ... , n) будем называть *альтернативой* правила вывода из цепочки α.

Пример грамматики:

$$G_{\text{example}} = \langle \{0, 1\}, \{A, S\}, P, S) \rangle,$$

где P состоит из правил:

P:

$S \rightarrow 0A1$
$0A \rightarrow 00A1$
$A \rightarrow \varepsilon$

Определение: цепочка $\beta \in (T \cup N)^*$ *непосредственно выводима* из цепочки $\alpha \in (T \cup N)^+$ в грамматике $G = \langle T, N, P, S \rangle$ (обозначим $\alpha \rightarrow_G \beta$), если $\alpha = \xi_1\gamma\xi_2$, $\beta = \xi_1\delta\xi_2$, где $\xi_1, \xi_2, \delta \in (T \cup N)^*$, $\gamma \in (T \cup N)^+$ и правило вывода $\gamma \rightarrow \delta$ содержится в P. Индекс G в обозначении $\rightarrow_G$ обычно опускают, если понятно, о какой грамматике идет речь.

Например, цепочка $00A11$ непосредственно выводима из $0A1$ в грамматике G_{example}: $\underline{0A}1 \rightarrow 0\underline{0A1}1$. Здесь цепочка $\underline{0A}$, подчеркнутая двойной чертой, играет роль подцепочки γ из определения, цепочка $\underline{00A1}$ играет роль подцепочки δ, $\xi_1 = \varepsilon$, $\xi_2 = 1$.

Определение: цепочка $\beta \in (T \cup N)^*$ *выводима* из цепочки $\alpha \in (T \cup N)^+$ в грамматике $G = \langle T, N, P, S \rangle$ (обозначим $\alpha \Rightarrow_G \beta$), если существуют цепочки $\gamma_0, \gamma_1, \ldots, \gamma_n$ ($n \geq 0$), такие, что $\alpha = \gamma_0 \rightarrow \gamma_1 \rightarrow \ldots \rightarrow \gamma_n = \beta$. Последовательность $\gamma_0, \gamma_1, \ldots, \gamma_n$ называется *выводом длины n*.

(Индекс G в обозначении $\Rightarrow_G$ опускают, если понятно, какая грамматика подразумевается.)

Например, $S \Rightarrow 000A111$ в грамматике G_{example} (см. пример выше), т.к. существует вывод $S \rightarrow 0A1 \rightarrow 00A11 \rightarrow 000A111$. Длина вывода равна 3.

Определение: *языком*, *порождаемым грамматикой* $G = \langle T, N, P, S \rangle$, называется множество $L(G) = \{\alpha \in T^* \mid S \Rightarrow \alpha\}$.

Другими словами, $L(G)$ — это все цепочки в алфавите T, которые выводимы из S с помощью правил P.

Например, $L(G_{\text{example}}) = \{0^n1^n \mid n > 0\}$.

Определение: цепочка $\alpha \in (T \cup N)^*$, для которой $S \Rightarrow \alpha$, называется *сентенциальной формой* в грамматике $G = \langle T, N, P, S \rangle$.

Таким образом, язык, порождаемый грамматикой, можно определить как множество терминальных сентенциальных форм.

Определение: грамматики G_1 и G_2 называются *эквивалентными*, если $L(G_1) = L(G_2)$.

Например, грамматики $G_1 = \langle \{0,1\}, \{A,S\}, P_1, S \rangle$ и $G_2 = \langle \{0,1\}, \{S\}, P_2, S \rangle$

P_1:
$S \to 0A1$
$0A \to 00A1$
$A \to \varepsilon$

P_2
$S \to 0S1 \mid 01$

эквивалентны, т. к. обе порождают язык $L(G_1) = L(G_2) = \{0^n1^n \mid n > 0\}$.

Определение: грамматики G_1 и G_2 *почти эквивалентны*, если $L(G_1) \cup \{\varepsilon\} = L(G_2) \cup \{\varepsilon\}$.

Другими словами, грамматики почти эквивалентны, если языки, ими порождаемые, отличаются не более, чем на ε.

Например,

$G_1 = \langle \{0,1\}, \{A,S\}, P_1, S \rangle$ и $G_2 = \langle \{0,1\}, \{S\}, P_2, S \rangle$

P_1:
$S \to 0A1$
$0A \to 00A1$
$A \to \varepsilon$

P_2
$S \to 0S1 \mid \varepsilon$

почти эквивалентны, т. к. $L(G_1) = \{0^n1^n \mid n > 0\}$, а $L(G_2) = \{0^n1^n \mid n \geq 0\}$, т. е. $L(G_2)$ состоит из всех цепочек языка $L(G_1)$ и пустой цепочки, которая в $L(G_1)$ не входит.

Классификация грамматик и языков по Хомскому

Определим с помощью ограничений на вид правил вывода четыре типа грамматик: тип 0, тип 1, тип 2, тип 3. Каждому типу грамматик соответствует свой класс[3)] языков.

Будем говорить, что язык, порождаемый грамматикой типа i (для $i=0, 1, 2, 3$), является языком *типа i*.

Тип 0

Любая порождающая грамматика является грамматикой *типа 0*. На вид правил грамматик этого типа не накладывается никаких дополнительных ограничений. Класс языков типа 0 совпадает с классом рекурсивно перечислимых языков.

Грамматики с ограничениями на вид правил вывода

Тип 1

Грамматика $G = \langle T, N, P, S \rangle$ называется *неукорачивающей*, если левая часть каждого правила из P не короче правой части (т. е. для любого правила $\alpha \to \beta \in P$ выполняется неравенство $|\alpha| \leq |\beta|$). В виде исключения в неукорачивающей грамматике допускается наличие правила $S \to \varepsilon$, при условии, что S (начальный символ) не встречается в правых частях правил.

Грамматикой *типа 1* будем называть неукорачивающую грамматику.

Тип 1 в некоторых источниках определяют с помощью так называемых контекстно-зависимых грамматик.

---

3) Классом обычно называют множество, элементы которого сами являются множествами.

Грамматика $G = \langle T, N, P, S \rangle$ называется *контекстно-зависимой* (*КЗ*), если каждое правило из P с непустой правой частью имеет вид $\alpha \to \beta$, где $\alpha = \xi_1 A \xi_2$, $\beta = \xi_1 \gamma \xi_2$, $A \in N$, $\gamma \in (T \cup N)^+$, $\xi_1, \xi_2 \in (T \cup N)^*$. В виде исключения в КЗ-грамматике допускается наличие правила $S \to \varepsilon$, при условии, что S (начальный символ) не встречается в правых частях правил.

Цепочку ξ_1 иногда называют *левым контекстом* (цепочку ξ_2 называют, соответственно, *правым контекстом*).

Язык, порождаемый контекстно-зависимой грамматикой, называется *контекстно-зависимым* языком.

Замечание. Из определений следует, что если язык, порождаемый контекстно-зависимой или неукорачивающей грамматикой $G = \langle T, N, P, S \rangle$, содержит пустую цепочку, то эта цепочка выводится в G за один шаг с помощью правила $S \to \varepsilon$. Других выводов для ε в грамматике G не существует.

Утверждение 1. Пусть L — формальный язык. Следующие утверждения эквивалентны:

- существует контекстно-зависимая грамматика G_1, такая что $L = L(G_1)$;
- существует неукорачивающая грамматика G_2, такая что $L = L(G_2)$.

Очевидно, что из (1) следует (2): любая контекстно-зависимая грамматика удовлетворяет ограничениям неукорачивающей грамматики (см. определения). Доказательство в обратную сторону основывается на том, что можно каждое неукорачивающее правило заменить эквивалентной серией контекстно-зависимых правил.

Из утверждения 1 следует, что язык, порождаемый неукорачивающей грамматикой, контекстно-зависим. Таким образом, неукорачивающие и КЗ-грамматики определяют один и тот же класс языков.

Тип 2

Грамматика $G = \langle T, N, P, S \rangle$ называется *контекстно-свободной* (*КС*), если каждое правило из P имеет вид $A \to \beta$, где $A \in N$, $\beta \in (T \cup N)^*$.

Заметим, что в КС-грамматиках допускаются правила с пустыми правыми частями.

Язык, порождаемый контекстно-свободной грамматикой, называется *контекстно-свободным* языком.

Грамматикой *типа 2* будем называть контекстно-свободную грамматику.

Утверждение 2. Для любой КС-грамматики G существует неукорачивающая КС-грамматика G', такая что $L(G) = L(G')$.

Согласно утверждению 2, любой контекстно-свободный язык порождается неукорачивающей контекстно-свободной грамматикой. Как следует из определений, такая КС-грамматика может содержать не более одного правила с пустой правой частью, причем это правило имеет вид $S \to \varepsilon$, где S — начальный символ, и при этом никакое правило из P не содержит S в своей правой части.

В разделе «Преобразования грамматик» будет приведен алгоритм позволяющий устранить из любой КС-грамматики все правила с пустыми правыми частями (в получившейся грамматике может быть правило $S \to \varepsilon$ в случае, если пустая цепочка при-

надлежит языку, при этом начальный символ S не будет входить в правые части правил).

Тип 3

Грамматика $G = \langle T, N, P, S \rangle$ называется *праволинейной*, если каждое правило из P имеет вид $A \to wB$ либо $A \to w$, где $A \in N, B \in N, w \in T^*$.

Грамматика $G = \langle T, N, P, S \rangle$ называется *леволинейной*, если каждое правило из P имеет вид $A \to Bw$ либо $A \to w$, где $A \in N, B \in N, w \in T^*$.

Утверждение 3. Пусть L — формальный язык. Следующие два утверждения эквивалентны:

- существует праволинейная грамматика G_1, такая что $L = L(G_1)$;
- существует леволинейная грамматика G_2, такая что $L = L(G_2)$.

Из утверждения 3 следует, что праволинейные и леволинейные грамматики определяют один и тот же класс языков. Такие языки называются *регулярными*. Право- и леволинейные грамматики также будем называть регулярными.

Регулярная грамматика является грамматикой *типа 3*.

Автоматной [4] грамматикой называется праволинейная (леволинейная) грамматика, такая, что каждое правило с непустой правой частью имеет вид: $A \to a$ либо $A \to aB$ (для леволинейной, соответственно, $A \to a$ либо $A \to Ba$), где $A \in N, B \in N$, $a \in T$.

Автоматная грамматика является более простой формой регулярной грамматики. Существует алгоритм, позволяющий по регулярной (право- или леволинейной) грамматике построить соответствующую автоматную грамматику. Таким образом, любой регулярный язык может быть порожден автоматной грамматикой. Кроме того, существует алгоритм, позволяющий устранить из регулярной (автоматной) грамматики все ε-альтернативы (кроме $S \to \varepsilon$ в случае, если пустая цепочка принадлежит языку; при этом S не будет встречаться в правых частях правил) [5]. Поэтому справедливо:

Утверждение 4. Для любой регулярной (автоматной) грамматики G существует неукорачивающая регулярная (автоматная) грамматика G', такая что $L(G) = L(G')$.

Иерархия Хомского

Утверждение 5. Справедливы следующие соотношения:

- любая регулярная грамматика является КС-грамматикой;
- любая неукорачивающая КС-грамматика является КЗ-грамматикой;
- любая неукорачивающая грамматика является грамматикой типа 0.

Утверждение 5 следует непосредственно из определений.

---

[4] В разделе «Разбор по регулярным грамматикам» будет рассмотрен алгоритм построения по конечному автомату эквивалентной грамматики. Построенная алгоритмом грамматика будет автоматной.

[5] Алгоритм устранения ε-альтернатив для КС-грамматик, приведенный в разделе «Преобразования грамматик», пригоден также и для устранения ε-альтернатив из регулярных и автоматных грамматик.

Рассматривая только неукорачивающие регулярные и неукорачивающие КС-грамматики, что согласно утверждениям 2 и 4 не ограничивает классы соответствующих языков, получаем следующую иерархию классов грамматик:

Неукорачивающие Регулярные $\subset$ *Неукорачивающие КС* $\subset$ *КЗ* $\subset$ *Тип 0* (Запись $A \subset B$ означает, что A является собственным подклассом класса B)

Замечение. Вместо класса КЗ-грамматик в данной иерархии можно указать более широкий класс неукорачивающих грамматик, которые, как известно, порождают те же языки, что и КЗ-грамматики.

Утверждение 6. Справедливы следующие соотношения:

- каждый регулярный язык является КС-языком, но существуют КС-языки, которые не являются регулярными, например:
$$L = \{a^n b^n \mid n > 0\};$$
- каждый КС-язык является КЗ-языком, но существуют КЗ-языки, которые не являются КС-языками, например:
$$L = \{a^n b^n c^n \mid n > 0\};$$
- каждый КЗ-язык является языком типа 0 (т. е. рекурсивно перечислимым языком), но существуют языки типа 0, которые не являются КЗ-языками, например: язык, состоящий из записей[6)] самоприменимых алгоритмов Маркова в некотором алфавите.

Из утверждения 6 следует иерархия классов языков:

Тип 3 (Регулярные) $\subset$ Тип 2 (КС) $\subset$ Тип 1 (КЗ) $\subset$ Тип 0

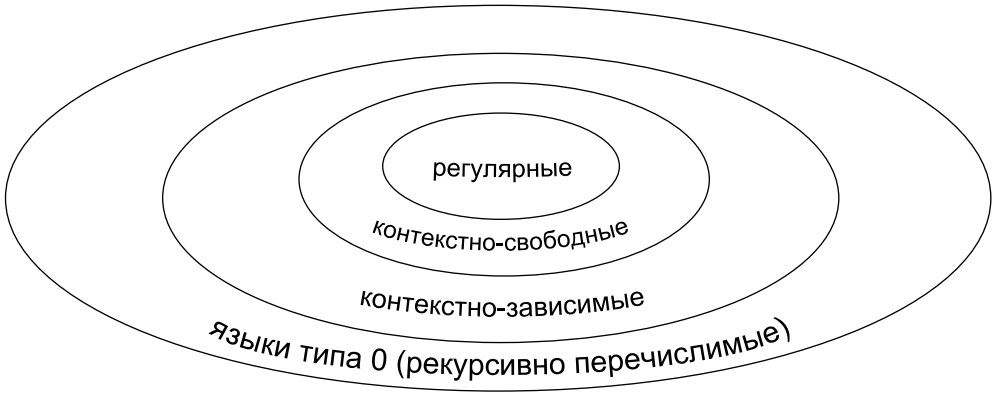


Рис. 1. Иерархия классов языков.

На рисунке 1 иерархия Хомского для классов языков изображена в виде диаграммы Венна. Классы вкладываются друг в друга. Самый широкий класс языков (типа 0) содержит в себе все остальные классы.

Утверждение 7. Проблема «Можно ли язык, описанный грамматикой типа k (k = 0, 1, 2), описать грамматикой типа k + 1 ?» является алгоритмически неразрешимой.

6 Понятие записи нормального алгоритма Маркова определяется в [10].

Заметим, что для $k = 1, 2, 3$ язык типа k является также и языком типа $k - 1$. Несмотря на то, что нет алгоритма, позволяющего по заданному описанию языка L (например, по грамматике), определить максимальное k, такое что «L является языком типа k», при ответе на вопрос «Какого типа заданный язык L ?» в нашем курсе будем указывать максимально возможное k (0, 1, 2, или 3) для заданного языка L.

Соглашение: в примерах и задачах для краткости вместо полной четверки $G = \langle T, N, P, S \rangle$ будем выписывать только так называемую «схему» грамматики — правила вывода P, считая, что большие латинские буквы обозначают нетерминальные символы, начальный символ (цель) грамматики обязательно стоит в левой части первого правила, ε — пустая цепочка, все остальные символы — терминальные.

Задача

Даны грамматики G_1 и G_2:

P_1:

$S \to 0A1$

$A \to 0A0$

$A \to \varepsilon$

P_2

$S \to aAb \mid \varepsilon$

(1) Какого типа язык $L(G_1)$? (2) Какого типа язык $L(G_2)$?

Решение

(1) Заметим, что правила грамматики G_1 удовлетворяют ограничениям на правила КС-грамматик, но не удовлетворяет ограничениям регулярных грамматик, т. е. это грамматика типа 2, и $L(G_1)$ является языком типа 2 (следовательно, это язык и типа 1, и типа 0). Нетрудно видеть, что $L(G_1)$ является также языком типа 3, поскольку описывается следующей регулярной грамматикой:

$S \to 0A$

$A \to 0B \mid 1$

$B \to 0A$

(2) По виду правил G_2 является грамматикой типа 2 и не является грамматикой типа 3. Следовательно, $L(G_2)$ является языком типа 2. Является ли $L(G_2)$ языком типа 3? Ответ на этот вопрос отрицательный, так как не существует регулярной грамматики (т. е. грамматики типа 3), порождающей язык $L(G_2) = \{a^n b^n \mid n \geq 0\}$ [7].

Примеры грамматик и языков

Примеры грамматик и языков, иллюстрирующие классы грамматик и языков, приведем в порядке, обратном классификации, т. е. будем идти «от простого к сложному».

Регулярные

1) Грамматика $S \to aS \mid a$ является праволинейной (неукорачивающей) грамматикой. Она порождает регулярный язык $\{a^n \mid n > 0\}$. Этот язык может быть порожден и леволинейной грамматикой: $S \to Sa \mid a$. Заметим, что обе грамматики из этого примера являются автоматными.

---

[7] Нерегулярность языка $\{a^n b^n \mid n \geq 0\}$ будет доказана в разделе «Разбор по регулярным грамматикам»

2) Грамматика $S \to aS \mid \varepsilon$ является праволинейной и порождает регулярный язык $\{a^n \mid n \geq 0\}$. Для любого регулярного языка существует неукорачивающая регулярная грамматика (см. утверждение 4). Построим неукорачивающую регулярную грамматику для данного языка:

$$S \to aA \mid a \mid \varepsilon$$
$$S \to a \mid aA$$

В самом деле, правило с пустой правой частью может применяться только один раз и только на первом шаге вывода, остальные правила таковы, что их правая часть не короче левой, т. е. грамматика неукорачивающая.

3) Грамматика

$$S \to A\perp \mid B\perp$$
$$A \to a \mid Ba$$
$$B \to b \mid Bb \mid A$$

леволинейная ; она порождает регулярный язык, состоящий из всех непустых цепочек в алфавите $\{a, b\}$, заканчивающихся символом $\perp$ (маркер конца) и не содержащих подцепочку aa. То есть в цепочках этого языка символ a не может встречаться два раза подряд, хотя бы один символ b обязательно присутствует между любыми двумя a. С помощью теоретико-множественной формулы данный язык описывается так: $\{\omega \perp \mid \omega \in \{a, b\}^+, aa \not\subset \omega\}$.

Контекстно-свободные

4) Грамматика

$$S \to aQb \mid accb$$
$$Q \to cSc$$

является контекстно-свободной (неукорачивающей) и порождает КС-язык $\{(ac)^n (cb)^n \mid n > 0\}$, который, как и встречавшийся нам ранее язык $\{a^n b^n \mid n \geq 0\}$, не является регулярным, т. е. не может быть порожден ни одной регулярной грамматикой.

5) Грамматика

$$S \to aSa \mid bSb \mid \varepsilon$$

порождает КС-язык $\{xx^R \mid x \in \{a,b\}^*\}$. Данный язык не является регулярным. Грамматика не удовлетворяет определению неукорачивающей, но для нее существует эквивалентная неукорачивающая грамматика (см. утверждение 2). Приведем такую грамматику:

$$S \to A \mid \varepsilon$$
$$A \to aAa \mid bAb \mid aa \mid bb$$

Неукорачивающие и контекстно-зависимые

6) Грамматика:

$$S \to aSBC \mid abC$$
$$CB \to BC$$
$$bB \to bb$$
$$bC \to bc$$
$$cC \to cc$$

неукорачивающая и порождает язык $\{a^n b^n c^n \mid n > 0\}$, который является языком типа 1, но не является языком типа 2 (этот факт доказывается с помощью «леммы о разрастании цепочек КС-языка» [3]).

Правило $CB \rightarrow BC$ не удовлетворяет определению КЗ-грамматики. Однако, заменив это правило тремя новыми $CB \rightarrow CD$, $CD \rightarrow BD$, $BD \rightarrow BC$ (добавляем новый символ D в множество нетерминалов N), мы получим эквивалентную серию контекстно-зависимых правил, которые меняют местами символы C и B. Таким образом, получаем эквивалентную КЗ-грамматику:

$S \rightarrow aSBC \mid abC$
$CB \rightarrow CD$
$CD \rightarrow BD$
$BD \rightarrow BC$
$bB \rightarrow bb$
$bC \rightarrow bc$
$cC \rightarrow cc$

Без ограничений на вид правил (тип 0)

7) В грамматике типа 0

$S \rightarrow SS$
$S \rightarrow \varepsilon$

второе правило не удовлетворяет ограничениям неукорачивающей грамматики. Существует бесконечно много выводов в данной грамматике, однако порождаемый язык конечен и состоит из единственной цепочки: $\{\varepsilon\}$.

Следующая грамматика также не является неукорачивающей и порождает пустой язык (для его обозначения используется «∅» — знак пустого множества), так как ни одна терминальная цепочка не выводится из S:

$S \rightarrow SS$
$S \rightarrow Sa \mid S$

Заметим, что языки $L_\varepsilon = \{\varepsilon\}$ и $L_\varnothing = \varnothing$ (пустой язык) могут быть описаны грамматиками типа 3. Кроме того, подчеркнем, что $L_\varepsilon \neq L_\varnothing$, т. к. $L_\varnothing$ не содержит цепочек вообще, а L_ε — язык, состоящий из одной цепочки (пустая цепочка ε по определению «равноправна» с остальными цепочками в алфавите).

8) Содержательные примеры грамматик, порождающих языки, не принадлежащие классу контекстно-зависимых (тип 1), не приводятся в нашем курсе из-за их громоздкости. Ниже обсуждается лишь возможность построения грамматики для одного языка типа 0.

Как известно, языки типа 0 (рекурсивно перечислимые множества) можно задавать с помощью распознавателей: НАМ или МТ. Существуют алгоритмы, позволяющие по НАМ или МТ построить эквивалентную грамматику типа 0, задающую тот же язык, что и исходный распознаватель.

Пусть задан некоторый алфавит. Из теории алгоритмов известно, что можно построить НАМ A, на вход которому подается запись любого алгоритма (НАМ) X в заданном алфавите, и алгоритм A эмулирует работу алгоритма X в применении к записи X. Можно также добиться, чтобы A зацикливался, если ему подали на вход цепочку, не являющуюся правильной записью какого-либо алгоритма. Таким образом, алгоритм A останавливается только на записях самоприменимых алгоритмов, а на всех других входах — зацикливается. Другими словами, A задает язык записей самоприменимых алгоритмов (обозначим этот язык L_{self}). Следовательно, L_{self} можно описать с помощью грамматики типа 0.

Покажем, что не существует грамматики типа 1, которая порождает язык L_{self}. Из теории алгоритмов известно, что проблема распознавания для грамматик типа 1 алгоритмически разрешима. То есть существует алгоритм, который определяет, принадле-

жит ли заданная цепочка грамматике. Предположим, существует грамматика типа 1, порождающая язык L_{self}. Тогда алгоритм распознавания дает ответ для любой цепочки, является ли она записью самоприменимого алгоритма или нет. Имеем противоречие с известным фактом об алгоритмической неразрешимости проблемы самоприменимости. Следовательно, грамматика типа 1, порождающая язык L_{self}, не существует.

Замечание о связи между языком и грамматикой

В приведенных ранее примерах мы утверждали, что данная грамматика порождает данный язык исходя из нестрогих рассуждений, интуитивно, по совокупности правил вывода. Вообще говоря, следует доказывать, что заданная грамматика порождает нужный язык. Для этого требуется доказать, что в данной грамматике:

а) выводится любая цепочка, принадлежащая языку,
б) не выводятся никакие другие цепочки.

Задача

Доказать, что грамматика G с правилами вывода:

P:

$S \to aSBC \mid abC$ (1,2)
$CB \to BC$ (3)
$bB \to bb$ (4)
$bC \to bc$ (5)
$cC \to cc$ (6)

порождает язык $L(G) = \{ a^n b^n c^n \mid n \geq 1 \}$.

(В скобках справа приведена нумерация для удобства ссылок на правила).

Решение

I) Приведем схемы порождения цепочек вида $a^n b^n c^n$, $n \geq 1$ с указанием номера правила на каждом шаге вывода.

Для $n = 1$: $S \to_{(2)} abC \to_{(5)} abc$

Для $n > 1$: $S \to_{(1)} aSBC \to_{(1)} aaSBCBC \to \ldots \to_{(1)} a^{n-1}S(BC)^{n-1} \to_{(2)} a^n bC(BC)^{n-1}$ $\to_{(3)} \ldots \to_{(3)} a^n bB^{n-1}C^n \to_{(4)} a^n bbB^{n-2}C^n \to \ldots \to_{(4)} a^n b^n C^n \to_{(5)} a^n b^n cC^{n-1} \to_{(6)}$ $a^n b^n ccC^{n-2} \to \ldots \to_{(6)} a^n b^n c^n$.

II) Из правил следует:

1. Новые символы a, $b \mid B$ и C появляются только при применении правил (1), (2) в равных количествах, т. е. в любой сентенциальной форме всегда **равное количество** a, $b \mid B$ и $c \mid C$.
2. Символ B заменяется только на b , а C — только на c.
3. Появившись, терминальные символы уже не меняют своей позиции, т. е. в любой сентенциальной форме символ ***a*** всегда **левее** любых ***b* | *B*** и ***c* | *C***.
4. Первый символ b появляется только после применения правила (2).
5. Символ B заменяется на b, только если слева от B стоит b, т. е. второй символ b появляется только непосредственно справа от первого b, третий b — непосредственно справа от второго b и т. д. Правило (5) применяется только после того, как исчерпана возможность применять (3), иначе вывод не будет завершен из-за наличия подцепочки cB в сентенциальной форме.

Из пунктов 3, 4, и 5 следует, что любой символ ***b*** расположен всегда **слева** от любого ***c* | *C***.

Следовательно, в любой выводимой цепочке равное количество a, b и c; a всегда стоит слева от b и c, b всегда стоит слева от c, т. е. любая цепочка имеет вид $a^n b^n c^n$, что и требовалось доказать.

Разбор цепочек

Цепочка принадлежит языку, порождаемому грамматикой, только в том случае, если существует ее вывод из начального символа этой грамматики. Процесс построения такого вывода (а, следовательно, и определения принадлежности цепочки языку) называется *разбором*[8]. Построение вывода можно осуществлять и в обратном порядке: в исходной цепочке ищем вхождение правой части некоторого правила и заменяем его на левую часть (это называется «сверткой»), в результате исходная цепочка «сворачивается» к некоторой сентенциальной форме, затем идет следующая свертка и т. д., пока не придем к цели грамматики — S.

С практической точки зрения наибольший интерес представляет разбор по **контекстно-свободным грамматикам.** Их порождающей мощности достаточно для описания большей части синтаксической структуры языков программирования, для различных подклассов КС-грамматик имеются хорошо разработанные практически приемлемые способы решения задачи разбора.

Рассмотрим основные понятия и определения, связанные с разбором по КС-грамматике.

Определение: вывод цепочки $\beta \in T^*$ из $S \in N$ в КС-грамматике $G = \langle T, N, P, S \rangle$, называется *левым (левосторонним)*, если в этом выводе каждая очередная сентенциальная форма получается из предыдущей заменой самого левого нетерминала.

Определение: вывод цепочки $\beta \in T^*$ из $S \in N$ в КС-грамматике $G = \langle T, N, P, S \rangle$, называется *правым (правосторонним)*, если в этом выводе каждая очередная сентенциальная форма получается из предыдущей заменой самого правого нетерминала.

В грамматике для одной и той же цепочки может быть несколько выводов, эквивалентных в том смысле, что в них в одних и тех же местах применяются одни и те же правила вывода, но в различном порядке.

Например, для цепочки $a + b + a$ в грамматике:

$$G_{expr} = \langle \{a, b, +\}, \{S, T\}, \{S \to T \mid T + S\,;\, T \to a \mid b\}, S \rangle$$

можно построить выводы:

(1) $S \to T + S \to T + T + S \to T + T + T \to a + T + T \to a + b + T \to a + b + a$

(2) $S \to T + S \to a + S \to a + T + S \to a + b + S \to a + b + T \to a + b + a$

(3) $S \to T + S \to T + T + S \to T + T + T \to T + T + a \to T + b + a \to a + b + a$

Здесь (2) — левосторонний вывод, (3) — правосторонний, а (1) не является ни левосторонним, ни правосторонним, но все эти выводы являются эквивалентными в указанном выше смысле.

---

[8] Разбором также называют и результат этого процесса, т. е. вывод цепочки, представленный (зафиксированный) каким-нибудь способом.

Для КС-грамматик можно ввести удобное графическое представление вывода, называемое деревом вывода, причем для всех эквивалентных выводов деревья вывода совпадают.

Определение: ориентированное упорядоченное[9] дерево называется *деревом вывода* (или *деревом разбора*) *в КС-грамматике* $G = \langle T, N, P, S \rangle$, если выполнены следующие условия:

(1) каждая вершина дерева помечена символом из множества $N \cup T \cup \{\varepsilon\}$, при этом корень дерева помечен символом S; листья — символами из $T \cup \{\varepsilon\}$;

(2) если вершина дерева помечена символом A, а ее непосредственные потомки — символами $a_1, a_2, \ldots, a_n$, где каждое $a_i \in T \cup N$, то $A \to a_1 a_2 \ldots a_n$ — правило вывода в этой грамматике;

(3) если вершина дерева помечена символом A, а ее единственный непосредственный потомок помечен символом ε, то $A \to \varepsilon$ — правило вывода в этой грамматике.

На рисунке 2 изображен пример дерева для цепочки $a + b + a$ в грамматике G_{expr}

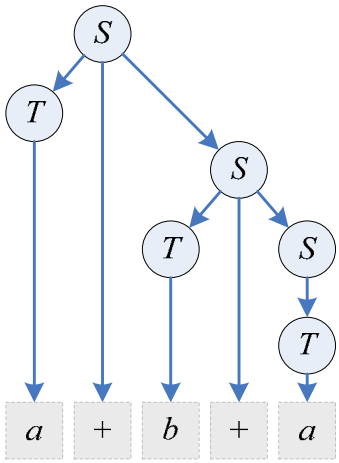


Рис. 2. Пример дерева вывода в грамматике G_{expr}.

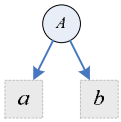
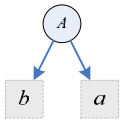
Определение: КС-грамматика G называется *неоднозначной*, если существует хотя бы одна цепочка $\alpha \in L(G)$, для которой может быть построено два или более различных деревьев вывода.

Это утверждение эквивалентно тому, что цепочка α имеет два или более разных левосторонних (или правосторонних) выводов.

Определение: в противном случае грамматика называется *однозначной*.

Определение: язык, порождаемый грамматикой, называется *неоднозначным*, если он не может быть порожден никакой однозначной грамматикой.

---

[9] Упорядоченность означает, что порядок расположения потомков вершины существен. Например, деревья  и  считаются различными:

Пример неоднозначной грамматики:

$$G_{if\text{-}then} = \langle \{\textbf{\textit{if}}, \textbf{\textit{then}}, \textbf{\textit{else}}, a, b\}, \{S\}, P, S \rangle,$$

где $P = \{S \to \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ S\ \textbf{\textit{else}}\ S \mid \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ S \mid a\}$.

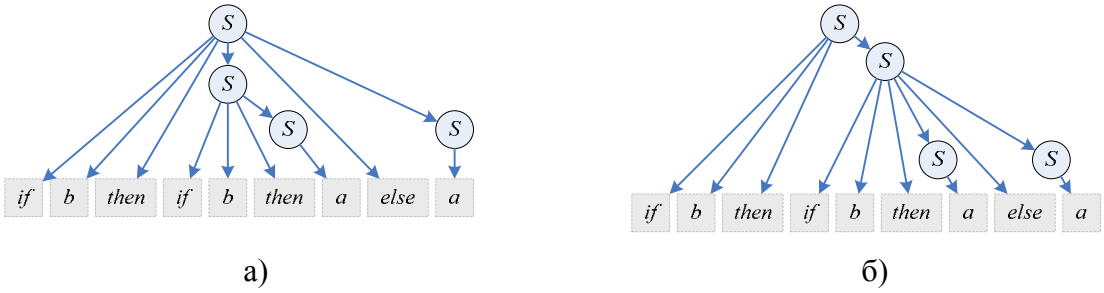
В этой грамматике для цепочки ***if*** *b* ***then if*** *b* ***then*** *a* ***else*** *a* можно построить два различных дерева вывода, изображенных на рисунке 3 (а, б).

Однако это не означает, что язык $L(G_{if\text{-}then})$ обязательно неоднозначный. Обнаруженная в $G_{if\text{-}then}$ неоднозначность — это свойство грамматики, а не языка. Для некоторых неоднозначных грамматик существуют эквивалентные им однозначные грамматики.

Если грамматика используется для определения языка программирования, то она должна быть однозначной.

В приведенном выше примере разные деревья вывода предполагают соответствие *else* разным *then*. Если договориться, что *else* должно соответствовать ближайшему к нему *then*, и подправить грамматику $G_{if\text{-}then}$, то неоднозначность будет устранена:

$S \quad \to \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ S \mid \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ S'\ \textbf{\textit{else}}\ S \mid a$
$S' \quad \to \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ S'\ \textbf{\textit{else}}\ S' \mid a$



а) б)

Рис. 3. Деревья вывода для «*if b then if b then a else a*» в грамматике $G_{if\text{-}then}$.

Утверждение 8. Проблема, порождает ли данная КС-грамматика однозначный язык (т. е. существует ли эквивалентная ей однозначная грамматика), является алгоритмически неразрешимой**.**

Более того, справедливо следующее утверждение:

Утверждение 9. Проблема, является ли данная КС-грамматика однозначной, алгоритмически неразрешима.

То есть процесс определения неоднозначна или однозначна заданная грамматика — это искусство поиска цепочки с двумя различными деревьями вывода, или доказательство, что таких цепочек не существует. Универсального способа, к сожалению, нет.

Однако можно указать некоторые виды правил вывода, которые приводят к неоднозначности (при условии, что эти правила не являются тупиковыми[10)], т. е. действительно используются на каком-нибудь шаге вывода терминальной цепочки из начального символа):

(в приводимых схемах $\alpha, \beta, \gamma \in (T \cup N)^*$)

(1) $A \to AA \mid \alpha$

[10)] Как избавиться от правил, не участвующих в построении выводов, см. в разделе «Преобразования грамматик»

(2) $A \to A\alpha A \mid \beta$

(3) $A \to \alpha A \mid A\beta \mid \gamma$ (здесь хотя бы одна из цепочек α или β не пуста)

(4) $A \to \alpha A \mid \alpha A \beta A \mid \gamma$

Отметим, что это всего лишь некоторые шаблоны. Все ситуации, приводящие к неоднозначности, перечислить невозможно в силу утверждения 9.

Пример неоднозначного КС-языка:

$$L = \{a^i b^j c^k \mid i = j \text{ или } j = k\}$$

Интуитивно это объясняется тем, что цепочки с $i = j$ должны порождаться группой правил вывода, отличных от правил, порождающих цепочки с $j = k$. Но тогда, по крайней мере, некоторые из цепочек с $i = j = k$ будут порождаться обеими группами правил и, следовательно, будут иметь по два разных дерева вывода. Доказательство того, что КС-язык L неоднозначный, приведено в [3, стр. 235–236].

Одна из грамматик, порождающих L, такова:

$S \to AB \mid DC$
$A \to aA \mid \varepsilon$
$B \to bBc \mid \varepsilon$
$C \to cC \mid \varepsilon$
$D \to aDb \mid \varepsilon$

Очевидно, что она неоднозначна.

Дерево вывода можно строить *нисходящим* либо *восходящим* способом.

При нисходящем разборе дерево вывода формируется от корня к листьям; на каждом шаге для вершины, помеченной нетерминальным символом, пытаются найти такое правило вывода, чтобы имеющиеся в нем терминальные символы «проецировались» на символы исходной цепочки.

Метод восходящего разбора на обратном построении вывода с помощью сверток от исходной цепочки к цели грамматики S. При этом дерево «растет» снизу вверх — от листьев (терминальных символов анализируемой цепочки) к корню S. Если грамматика однозначная, то при любом способе построения будет получено одно и то же дерево разбора.

Преобразования грамматик

В некоторых случаях КС-грамматика может содержать бесполезные символы, которые не участвуют в порождении цепочек языка и поэтому могут быть удалены из грамматики.

Определение: символ $x \in T \cup N$ называется *недостижимым* в грамматике $G = \langle T, N, P, S \rangle$, если он не появляется ни в одной сентенциальной форме этой грамматики.

Алгоритм удаления недостижимых символов

Вход: КС-грамматика $G = \langle T, N, P, S \rangle$,

Выход: КС-грамматика $G' = \langle T', N', P', S \rangle$, не содержащая недостижимых символов, для которой $L(G) = L(G')$.

Метод:

$V_0 := \{S\}$; $i := 1$.

$V_i := \{x \mid x \in T \cup N$, в P есть $A \to \alpha x \beta$ и $A \in V_{i-1}$, $\alpha, \beta \in (T \cup N)^*\} \cup V_{i-1}$.

Если $V_i \neq V_{i-1}$, то $i := i + 1$ и переходим к шагу 2, иначе $N' := V_i \cap N$; $T' := V_i \cap T$; P' состоит из правил множества P, содержащих только символы из V_i; $G' := \langle T', N', P', S \rangle$.

Определение: символ $A \in N$ называется *бесплодным* в грамматике $G = \langle T, N, P, S \rangle$, если множество $\{\alpha \in T^* \mid A \Rightarrow \alpha\}$ пусто.

Алгоритм удаления бесплодных символов

Вход: КС-грамматика $G = \langle T, N, P, S \rangle$.

Выход: КС-грамматика $G' = \langle T', N', P', S \rangle$, не содержащая бесплодных символов, для которой $L(G) = L(G')$.

Метод:

Строим множества $N_0, N_1, \ldots$

$N_0 := \varnothing$, $i := 1$.

$N_i := \{A \mid (A \to \alpha) \in P$ и $\alpha \in (N_{i-1} \cup T)^*\} \cup N_{i-1}$.

Если $N_i \neq N_{i-1}$, то $i := i + 1$ и переходим к шагу 2, иначе $N' := N_i$; P' состоит из правил множества P, содержащих только символы из $N' \cup T$; $G' := \langle T', N', P', S \rangle$.

Определение: КС-грамматика G называется *приведенной,* если в ней нет недостижимых и бесплодных символов.

Алгоритм приведения грамматики

Обнаруживаются и удаляются все бесплодные нетерминалы.

Обнаруживаются и удаляются все недостижимые символы.

Удаление символов сопровождается удалением правил вывода, содержащих эти символы.

Замечание: если в этом алгоритме переставить шаги (1) и (2), то не всегда результатом будет приведенная грамматика.

Для описания синтаксиса языков программирования стараются использовать однозначные приведенные КС-грамматики.

Некоторые применяемые на практике алгоритмы разбора по КС-грамматикам требуют, чтобы в грамматиках не было правил с пустой правой частью, т. е. чтобы КС-грамматика была неукорочивающей. Для любой КС-грамматики существует эквивалентная неукорачивающая (см. утверждение 2).

Ниже приводится алгоритм, позволяющий преобразовать любую КС-грамматику в неукорачивающую. На первом шаге алгоритма строится множество X, состоящее из нетерминалов грамматики, из которых выводима пустая цепочка. Построение этого множества можно провести по аналогии с шагами алгоритма удаления

бесплодных символов. На первом шаге: $X_1 := \{ A \mid (A \to \varepsilon) \in P \}$; i :=2. На втором шаге: $X_i := \{A \mid (A \to \alpha) \in P$ и $\alpha \in X_{i-1}^*\} \cup X_{i-1}$. Далее, пока $X_i \neq X_{i-1}$ увеличиваем i на единицу и повторяем второй шаг.

Алгоритм устранения правил с пустой правой частью

Вход: КС-грамматика $G = \langle T, N, P, S \rangle$.

Выход: КС-грамматика $G' = \langle T, N', P', S' \rangle$, G' — неукорачивающая, $L(G') = L(G)$.

Метод:

1. Построить множество $X = \{A \in N \mid A \Rightarrow \varepsilon\}$; $N' := N$.

2. Построить P', удалив из множества правил P все правила с пустой правой частью.

3. Если $S \in X$, то ввести новый начальный символ S', добавив его в N', и в множество правил P' добавить правило $S' \to S \mid \varepsilon$. Иначе просто переименовать S в S'.

4. Изменять P' следующим образом. Для любого $A \in X$ правило вида $B \to \alpha_1 A \alpha_2 A ... \alpha_n A \alpha_{n+1}$, где $\alpha_i \in (\ (N' - \{A\}) \cup T\ )^*$, заменить 2^n правилами, соответствующими всем возможным комбинациям вхождений A между α_i:

$B \to \alpha_1 \alpha_2 ... \alpha_n \alpha_{n+1}$

$B \to \alpha_1 \alpha_2 ... \alpha_n A \alpha_{n+1}$

...

$B \to \alpha_1 \alpha_2 A ... \alpha_n A \alpha_{n+1}$

$B \to \alpha_1 A \alpha_2 A ... \alpha_n A \alpha_{n+1}$

Замечание: если $\alpha_i = \varepsilon$ для всех $i = 1,..., n+1$, то получившееся на данном шаге правило $B \to \varepsilon$ не включать в множество P'. В результате этого шага в P' не останется правил с нетерминалами из X.

5. Удалить бесплодные и недостижимые символы и правила, их содержащие. (Кроме изначально имеющихся (в неприведенной грамматике),

бесполезные символы могут возникнуть в результате шагов 2–4).

Замечание. Если применить данный алгоритм к регулярной (автоматной) грамматике, то результатом будет неукорачивающая регулярная (автоматная) грамматика.

Далее везде, если не оговорено иное, будем рассматривать только приведенные грамматики.

Элементы теории трансляции

Введение

В этом разделе будут рассмотрены некоторые алгоритмы и технические приемы, применяемые при построении трансляторов. Практически во всех трансляторах (и в компиляторах, и в интерпретаторах) в том или ином виде присутствует большая часть перечисленных ниже процессов:

- лексический анализ,
- синтаксический анализ,
- семантический анализ,
- генерация внутреннего представления программы,
- оптимизация,
- генерация объектной программы.

В конкретных компиляторах порядок этих процессов может быть несколько иным, какие-то из них могут объединяться в одну фазу, другие могут выполняться в течение всего процесса компиляции. В интерпретаторах и при смешанной стратегии трансляции часть этих процессов может вообще отсутствовать.

В данном разделе мы рассмотрим некоторые методы, используемые для построения анализаторов (лексического, синтаксического и семантического), язык промежуточного представления программы, способ генерации промежуточной программы, ее интерпретации. Излагаемые алгоритмы и методы иллюстрируются на примере модельного паскалеподобного языка (М-языка). Приводится реализация в виде программы на Си++.

Информацию о других методах, алгоритмах и приемах, используемых при создании трансляторов, можно найти в [1, 2, 3, 4, 5, 8].

Разбор по регулярным грамматикам

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[11)]. Напомним, что в такой грамматике каждое правило из P имеет вид $A \to Bt$ либо $A \to t$, где $A \in N$, $B \in N$, $t \in T$.

Соглашение: предположим, что анализируемая цепочка заканчивается специальным символом «⊥» — *признаком конца цепочки*.

[11)] Полное отсутствие ε-правил в грамматике не позволяет описывать языки, содержащие пустую цепочку. Для наших целей в данном разделе это ограничение оправдано — мы будем применять автоматные грамматики для описания и разбора лексических единиц (лексем) языков программирования. Лексемы никогда не бывают пустыми.

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

(1) 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)

(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 $B \rightarrow Aa_i$ (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 $a_1a_2...a_n\perp \notin L(G)$.

(4) 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.* Анализ этой ситуации будет дан ниже.

Допустим, что разбор на каждом шаге детерминированный.

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).

Это можно сделать в виде таблицы, строки которой помечены нетерминальными символами грамматики, столбцы — терминальными. Значение каждого элемента таблицы — это нетерминальный символ, к которому можно свернуть пару «нетерминал-терминал», которыми помечены соответствующие строка и столбец.

Например, для леволинейной грамматики $G_{left} = \langle \{a, b, \perp\}, \{S, A, B, C\}, P, S \rangle$, такая таблица будет выглядеть следующим образом:

P:

$S \rightarrow C\perp$
$C \rightarrow Ab \mid Ba$
$A \rightarrow a \mid Ca$
$B \rightarrow b \mid Cb$

	a	b	$\perp$
C	A	B	S
A	–	C	–
B	C	–	–
S	–	–	–

Знак «−» ставится в том случае, если для пары «терминал-нетерминал» свертки нет.

Но чаще информацию о возможных свертках представляют в виде *диаграммы состояний (ДС)* — неупорядоченного ориентированного помеченного графа, который строится следующим образом:

(1) строят 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} (см. пример выше):

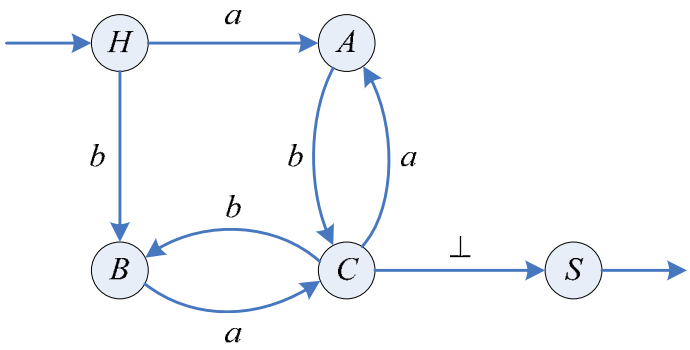


Рис. 4. Диаграмма состояний для грамматики G_{left}.

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 $L(G)$.

4) 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*. Анализ этой ситуации будет приведен ниже.

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: *детерминированный конечный автомат* (*ДКА*) — 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 K, определяющее поведение автомата;

$H \in K$ — начальное состояние;

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

Замечания к определению ДКА:

(1) Заключительных состояний в ДКА может быть более одного, однако для любого регулярного языка, все цепочки которого заканчиваются маркером конца ($\perp$), существует ДКА с единственным заключительным состоянием. Заметим также, что ДКА, построенный по регулярной грамматике рассмотренным выше способом, всегда будет иметь единственное заключительное состояние S [12)].

(2) Отображение δ: $K \times T \to K$ 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».

[12)] Нетрудно указать и обратный способ — построение грамматики по диаграмме состояний автомата, — причем получившаяся грамматика будет автоматной. Каждой дуге из начального состояния H в состояние W, помеченной символом t, будет соответствовать правило $W \to t$, каждой дуге из состояния V в состояние W, помеченной символом t, будет соответствовать правило $W \to Vt$. Заключительное состояние S объявляется начальным символом грамматики.

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

P:

$S \rightarrow C\perp$
$C \rightarrow Ab \mid Ba$
$A \rightarrow a \mid Ca$
$B \rightarrow b \mid Cb$

анализатор будет таким:

```
#include <iostream.h>

char c;

void gc ()
{
    cin >> c;
}

bool scan_G ()
{
    enum state { H, A, B, C, S, ER };  // множество состояний
    state CS;                          // CS -- текущее состояние

    CS = H;
    gc ();
    do
    {
        switch (CS)
        {
        case H:  if ( c == 'a' )
                 {
                     gc ();
                     CS = A;
                 }
                 else if ( c == 'b' )
                 {
                     gc ();
                     CS = B;
                 }
```

```
                    else
                        CS = ER;
                    break;
            case A: if ( c == 'b' )
                    {
                        gc();
                        CS = C;
                    }
                    else
                        CS = ER;
                    break;
            case B: if ( c == 'a' )
                    {
                        gc();
                        CS = C;
                    }
                    else
                        CS = ER;
                    break;
            case C: if ( c =='a' )
                    {
                        gc();
                        CS = A;
                    }
                    else if ( c == 'b' )
                    {
                        gc();
                        CS = B;
                    }
                    else if ( c == '⊥' )
                        CS = S;
                    else
                        CS = ER;
                    break;
            }
        }
        while ( CS != S && CS != ER );
        if ( CS == ER )
            return false;
        else
            return true;
    }
```

Пример разбора цепочки

Рассмотрим работу анализатора для грамматики G 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 $L \to Nt$ 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:

$$\underline{a}bba\perp \leftarrow \underline{Ab}ba\perp \leftarrow \underline{Cb}a\perp \leftarrow \underline{Ba}\perp \leftarrow \underline{C\perp} \leftarrow S$$

Эта последовательность не что иное, как обращение (правого) вывода цепочки $abba\perp$ в грамматике G. Она соответствует построению дерева снизу вверх (см. рис. 5):

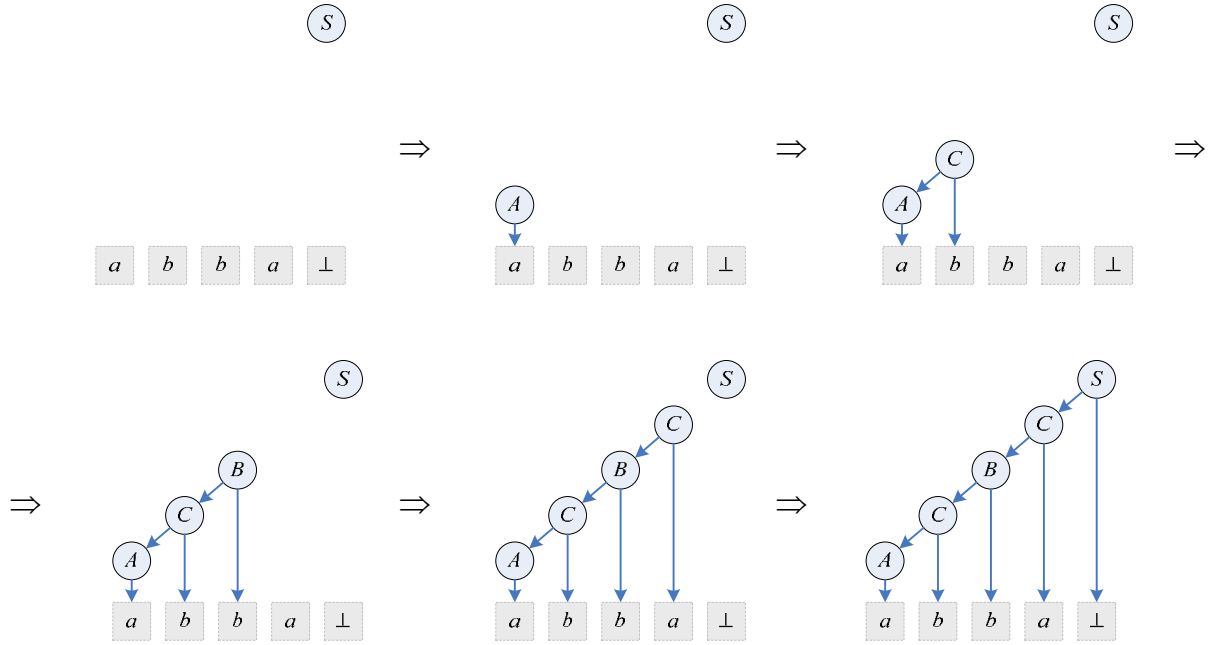


Рис. 5. Построение дерева вывода сверху вниз.

Разбор по праволинейной грамматике

По диаграмме состояний (см. рис. Рис. 4) 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[13)].

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

P:

$$\begin{aligned} H &\to aA \mid bB \\ A &\to bC \\ C &\to bB \mid aA \mid \perp \\ B &\to aC \end{aligned}$$

Заметим, что $L(G_{right}) = L(G_{left})$, так как грамматики G_{right} и G_{left} соответствуют одной и той же ДС (см. рис. Рис. 4).

Рассмотрим разбор цепочки $abba\perp$ по праволинейной грамматике G_{right}. Последовательность переход в ДС для этой цепочки такова:

---

13) Нетрудно предложить 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 $V \to t\,W$ проводится дуга из V в W, помеченная символом t. Начальным состоянием в ДС будет начальный символ H .

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

Каждый переход, за исключением последнего, означает теперь замену в сентенциальной форме нетерминала на пару «терминал-нетерминал» с помощью некоторого правила вывода грамматики G_{right}. В результате получаем следующий (левый) вывод, который соответствует последовательности переходов в ДС:

$$H \to \underline{aA} \to a\underline{bC} \to ab\underline{bB} \to abb\underline{aC} \to abba\underline{\perp}$$

Такой вывод отражает построение дерева вывода сверху вниз (см. рис. 6):

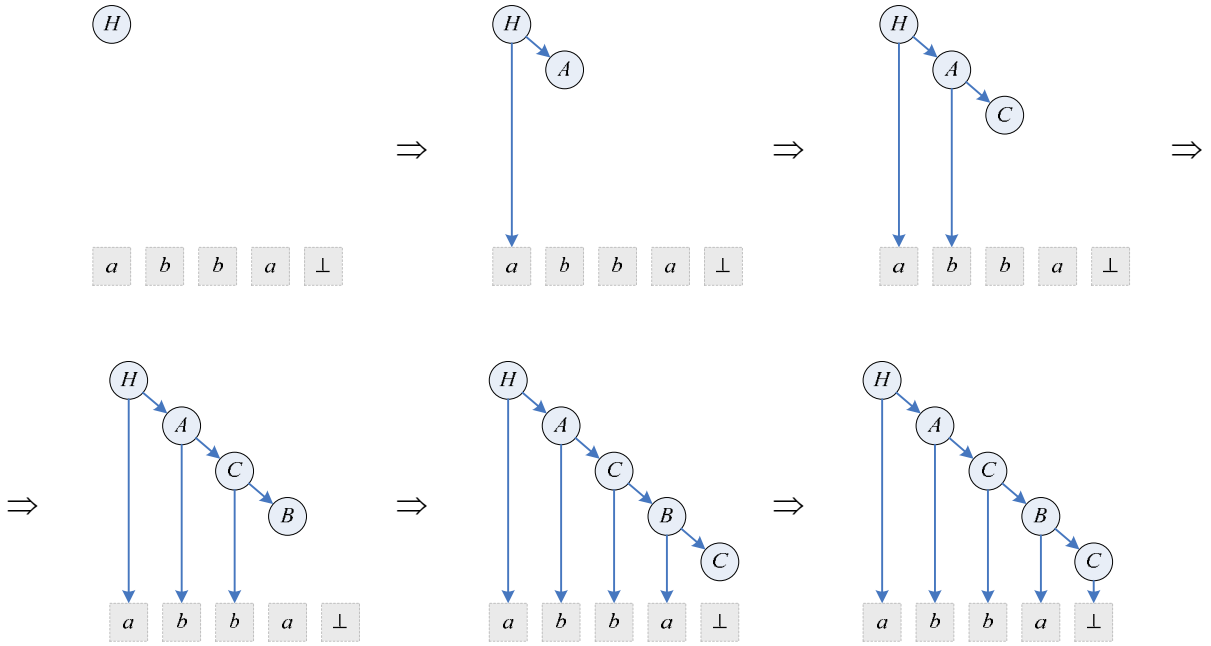


Рис. 6. Построение дерева вывода сверху вниз.

О недетерминированном разборе

При анализе по регулярной грамматике может оказаться, что несколько нетерминалов имеют одинаковые правые части, и поэтому неясно, к какому из них делать свертку (см. ситуацию 4 в описании алгоритма). В терминах диаграммы состояний это означает, что из одного состояния выходит несколько дуг, ведущих в разные состояния, но помеченных одним и тем же символом.

Например, 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.

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;

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

$S \subset K$ — 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, $i = 1, 2, ..., n$. Также как и ДКА, любой НКА можно представить в виде таблицы (в одной ячейке такой таблицы можно указывать сразу несколько состояний, в которые возможен переход по 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 $B \in \delta(A, t)$.

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 НКА одновременно является и заключительным, то НКА допускает пустую цепочку ε. В данном разделе мы рассматриваем автоматы, построенные по регулярным грамматикам без пустых правых частей, поэтому цепочка ε не может допускаться соответствующим автоматом.

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».

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 10. 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 $\delta(C, a) = b$ эквивалентный переход в НКА $\delta(C, a) = \{b\}$, начальное состояние ДКА поместить в множество начальных состояний НКА, а заключительное состояние ДКА поместить в множество заключительных состояний НКА. Приводимый ниже алгоритм построения ДКА, эквивалентного НКА, обосновывает то, что из (3) следует (2).

Алгоритм построения ДКА по НКА

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

Выход: ДКА $M_1 = \langle K_1, T, \delta_1, H_1, S_1 \rangle$, 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 боле 2^s элементов , где s — мощность K.

Подмножество $\{A_1, A_2, \dots, A_n\}$ состояний из K будем для краткости записывать как $\underline{A_1A_2...A_n}$. Множество K_1 и переходы, определяющие функцию δ_1, будем строить, начиная с состояния H_1: $H_1 := \underline{A_1A_2...A_n}$, где $A_1, A_2, \dots, A_n \in H$. 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 δ_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 $\delta_1 (\underline{A_1A_2...A_m}, t) = \underline{B_1B_2...B_k}$, где для $1 \le j \le k$ в НКА $\delta (A_i, t) = B_j$ 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 $\delta_1 (\underline{A_1A_2...A_m}, t) = \varnothing$).
- Добавляем в K_1 новое состояние $\underline{B_1B_2...B_k}$.

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: $S_1 := \{\underline{A_1A_2...A_m} \mid \underline{A_1A_2...A_m} \in K_1 , A_i \in S$ для некоторых $1 \le i \le m\}$.

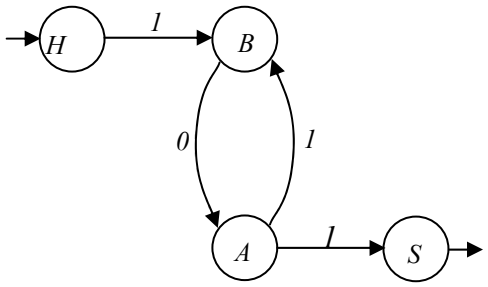
Замечание. Множество S_1 построенного ДКА может состоять более, чем из одного элемента. Не для всех регулярных языков существует эквивалентный ДКА с единственным заключительным состоянием (пример — язык всех цепочек в алфавите $\{a, b\}$, содержащих не более двух символов b). Однако для реализации алгоритма детерминированного разбора заключительное состояние должно быть единственным. В таком случае изменяют входной язык, добавляя маркер «⊥» в конец каждой цепочки (на практике в роли маркера конца цепочки ⊥ может выступать признак конца файла, признак конца строки или другие разделители). Вводится новое состояние S, и для каждого состояния Q из множества S_1 добавляется переход по символу ⊥: $\delta_1 (\underline{Q}, \perp) = S$. 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. Задан НКА $M = \langle \{ H, A, B, S \}, \{0, 1\}, \delta, \{H\}, \{S\} \rangle$, где

$\delta (H, 1) = \{B\}$ ДС для M :
$\delta (A, 1) = \{B, S\}$

$\delta (B, 0) = A$



$L(M) = \{ 1(01)^n \mid n \geq 1 \}$

Грамматика, соответствующая M:

$S \rightarrow A1$
$A \rightarrow B0$
$B \rightarrow A1 \mid 1$

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

$\delta_1(\underline{H}, 1) = \underline{B}$

$\delta_1(\underline{B}, 0) = \underline{A}$

$\delta_1(\underline{A}, 1) = \underline{BS}$

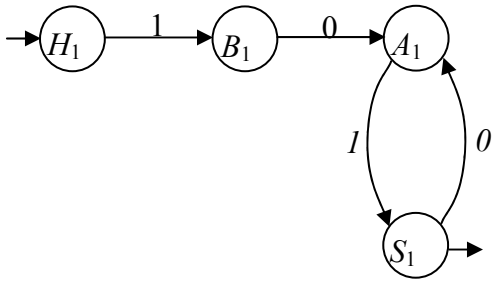
$\delta_1(\underline{BS}, 0) = \underline{A}$

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

Таким образом, $M_1 = \langle \{\underline{H}, \underline{B}, \underline{A}, \underline{BS}\}, \{0, 1\}, \delta_1, \underline{H}, \underline{BS} \rangle$. Для удобства переименуем в M_1 состояния так, что $\underline{BS}$ обозначается теперь как S_1, а в однобуквенных именах состояний вместо подчеркивания используется индекс 1. Тогда $M_1 = \langle \{H_1, B_1, A_1, S_1\}, \{0, 1\}, \{ \delta_1(H_1, 1) = B_1; \delta_1(B_1, 0) = A_1; \delta_1(A_1, 1) = S_1; \delta_1(S_1, 0) = A_1\}, H_1, S_1 \rangle$.

Грамматика и ДС, соответствующие M_1:

$S_1 \rightarrow A_1 1$
$A_1 \rightarrow S_1 0 \mid B_1 0$
$B_1 \rightarrow 1$



Пример 2. Задана 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 эквивалентную грамматику, по которой возможен детерминированный разбор.

Решение

Построим функцию переходов НКА:

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

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

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

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

Процесс построения функции переходов ДКА удобно изо-бразить в виде таблицы, начав ее заполнение с начального состояния Н и затем добавляя строки для вновь появляющихся состояний: Переходы ДКА:

символ / состояние	***a***	***b***

$\underline{H}$	$\underline{S}$	$\underline{A}$
$\underline{S}$	$\varnothing$	$\underline{AS}$
$\underline{A}$	$\underline{AS}$	$\varnothing$
$\underline{AS}$	$\underline{AS}$	$\underline{AS}$

$\delta_1(\underline{H}, a) = \underline{S}$

$\delta_1(\underline{H}, b) = \underline{A} \Rightarrow \delta_1(\underline{S}, b) = \underline{AS}$

$\delta_1(\underline{A}, a) = \underline{AS}$

$\delta_1(\underline{AS}, a) = \underline{AS}$

$\delta_1(\underline{AS}, b) = \underline{AS}$

Обозначим $\underline{A}$ через A, $\underline{H} \Rightarrow H$, $\underline{AS} \Rightarrow Y$, а $\underline{S} \Rightarrow X$. 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:

$\delta_1(H, a) = X$

$\delta_1(H, b) = A$

$\delta_1(X, b) = Y$

$\delta_1(X, \perp) = S$

$\delta_1(A, a) = Y$

$\delta_1(Y, a) = Y$

$\delta_1(Y, b) = Y$

$\delta_1(Y, \perp) = S$

Получим следующую грамматику G_1, позволяющую воспользоваться алгоритмом детерминированного разбора

G_1:

$S \rightarrow X\perp \mid Y\perp$

$Y \rightarrow Ya \mid Yb \mid Aa \mid Xb$

$X \rightarrow a$

$A \rightarrow b$

В заключение раздела о регулярных языках приведем пример использования автоматов в решении теоретических задач.

Задача.

Доказать, что контекстно-свободный язык $L = \{a^n b^n \mid n \geq 1\}$ нерегулярен.

Доказательство (от противного). Предположим, что язык L регулярен. Тогда существует конечный автомат (ДКА или НКА) $A = (K, \Sigma, \delta, I, F)$, допускающий язык L: $L(A) = L$. (Согласно утверждению 10, любой регулярный язык может быть задан конечным автоматом). Пусть число состояний автомата A равно k ($k > 0$). Рассмотрим цепочку $a^k b^k \in L$. Так как $L = L(A)$, $a^k b^k \in L(A)$. Это означает, что в автомате A есть успешный путь T с пометкой $a^k b^k$:

$$p_1 \xrightarrow{a} p_2 \xrightarrow{a} \cdots \xrightarrow{a} p_k \xrightarrow{a} p_{k+1} \xrightarrow{b} p_{k+2} \xrightarrow{b} \cdots \xrightarrow{b} p_{2k} \xrightarrow{b} p_{2k+1},$$

где $p_i \in K$ *для* $i = 1,\ldots,2k+1$. Так как в автомате A всего k состояний, среди p_1, p_2, …, p_{k+1} найдутся хотя бы два одинаковых. Иными словами, существуют i, j, $1 \le i < j \le k$, такие что $p_i = p_j$. Таким образом, участок $p_i \xrightarrow{a} \cdots \xrightarrow{a} p_j$ пути T является циклом. Пусть длина этого цикла равна l ($l > 0$, так как цикл — это непустой путь). Рассмотрим успешный путь T', который отличается от T тем, что циклический участок $p_i \xrightarrow{a} \cdots \xrightarrow{a} p_j$ присутствует в нем дважды:

$$p_1 \xrightarrow{a} \cdots p_i \xrightarrow{a} \cdots \xrightarrow{a} (p_i = p_j) \xrightarrow{a} \cdots \xrightarrow{a} p_j \xrightarrow{a} \cdots p_{k+1} \xrightarrow{b} \cdots p_{2k+1} \ .$$

Пометкой пути T' является цепочка $a^{k+l}b^k$. Поскольку T' — успешный путь, $a^{k+l}b^k \in L(A)$. Так как $a^{k+l}b^k \notin L$, получаем, что $L \ne L(A)$. Это противоречит равенству $L = L(A)$. Следовательно, предположение о том, что L регулярен, неверно □

Задачи лексического анализа

Лексический анализ (ЛА) — это первый этап процесса компиляции. На этом этапе литеры, составляющие исходную программу, группируются в отдельные лексические элементы, называемые *лексемами*.

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.

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).

Соглашение: эту пару тоже будем называть лексемой, если это не будет вызывать недоразумений.

Таким образом, лексический анализатор — это транслятор, входом которого служит цепочка символов, представляющих исходную программу, а выходом — последовательность лексем.

Очевидно, что лексемы перечисленных выше типов можно описать с помощью регулярных грамматик. Поскольку для языков программирования понятия пустой цепочки не существует, для описания лексического состава любого языка программирования достаточно автоматных грамматик без ε-правил.

Например, идентификатор (I):

$I \to a \mid b \mid \ldots \mid z \mid Ia \mid Ib \mid \ldots \mid Iz \mid I0 \mid I1 \mid \ldots \mid I9$

целое без знака (N):

$N \to 0 \mid 1 \mid \ldots \mid 9 \mid N0 \mid N1 \mid \ldots \mid N9$

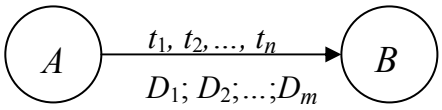
и т. д.

Для грамматик этого класса, как мы уже видели, существует простой и эффективный алгоритм анализа того, принадлежит ли заданная цепочка языку, порождаемо-

му 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 того, чтобы решить эту задачу, опираясь на способ анализа 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 $D_1, D_2, \ldots, D_m$.

Лексический анализатор для М-языка

Описание модельного языка

$P \rightarrow$ ***program*** D_1; $B\perp$
$D_1 \rightarrow$ ***var*** D {,D}
$D \rightarrow I$ {, I}: [***int*** | ***bool***]
$B \rightarrow$ ***begin*** S {;S} ***end***
$S \rightarrow I := E$ | ***if*** E ***then*** S ***else*** S | ***while*** E ***do*** S | B | ***read*** (I) | ***write*** (E)
$E \rightarrow E_1$ [= | < | > | !=] E_1 | E_1
$E_1 \rightarrow T$ {[+ | − | ***or***] T}
$T \rightarrow F$ {[∗ | / | ***and***] F}
$F \rightarrow I$ | N | L | ***not*** F | (E)
$L \rightarrow$ ***true*** | ***false***
$I \rightarrow C$ | IC | IR
$N \rightarrow R$ | NR
$C \rightarrow a$ | b | ... | z | A | B | ... | Z
$R \rightarrow 0$ | 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 αα, либо ααα 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 ⊥ — 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 «⊥» ⟩ }.

true, ***false***, ***read*** и ***write*** — служебные слова (их нельзя переопределять, как стандартные идентификаторы Паскаля).

Сохраняется паскалевское правило о разделителях между идентификаторами, числами и служебными словами.

Итак, перейдем к лексическому анализатору.

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.

Лексический анализатор для модельного языка будем писать в несколько этапов: сначала опишем на С++ классы объектов, которые затем будем использовать при ЛА, затем построим ДС с действиями для распознавания и формирования внутреннего представления лексем, а затем по ДС напишем программу ЛА. Отметим, что мы будем рассматривать один из возможных способов проектирования и реализации программы ЛА на С++, быть может, не самый лучший.

Проектирование классовой структуры лексического анализатора

Представление лексем: выделим следующие типы лексем, введя следующий перечислимый тип данных:

```
enum type_of_lex
{
    LEX_NULL,                                  // 0
    LEX_AND, LEX_BEGIN, … LEX_WRITE,           // 18
    LEX_FIN,                                   // 19
    LEX_SEMICOLON, LEX_COMMA, … LEX_GEQ,       // 35
    LEX_NUM,                                   // 36
    LEX_ID,                                    // 37
    POLIZ_LABEL,                               // 38
    POLIZ_ADDRESS,                             // 39
    POLIZ_GO,                                  // 40
    POLIZ_FGO                                  // 41
};
```

Содержательно внутреннее представление лексем — это пара (тип_лексемы, значение_лексемы). Значение лексемы — это номер строки в таблице лексем соответствующего класса, содержащей информацию о лексеме, или непосредственное значение, например, в случае целых чисел.

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.

Итак, класс *Lex*:

```
class Lex
{
    type_of_lex  t_lex;
    int          v_lex;
public:
    Lex ( type_of_lex t = LEX_NULL, int v = 0)
    {
        t_lex = t; v_lex = v;
    }
    type_of_lex  get_type () { return t_lex; }
    int get_value () { return v_lex; }
    friend ostream& operator << (ostream &s, Lex l)
    {
        s << '(' << l.t_lex << ',' << l.v_lex << ");";
        return s;
    }
};
```

Класс *Ident*:

```
class Ident
{
    char     *   name;
    bool         declare;
    type_of_lex  type;
    bool         assign;
    int          value;
public:
    Ident ()
    {
        declare  = false;
        assign   = false;
    }
    char *get_name ()
    {
        return name;
    }
```

```
        void put_name (const char *n)
        {
            name = new char [ strlen(n) + 1 ];
            strcpy ( name, n );
        }
        bool get_declare ()
        {
            return declare;
        }
        void put_declare ()
        {
            declare = true;
        }
        type_of_lex get_type ()
        {
            return type;
        }
        void put_type ( kind_of_lex t )
        {
            type = t;
        }
        bool get_assign ()
        {
            return assign;
        }
        void put_assign ()
        {
            assign = true;
        }
        int get_value ()
        {
            return value;
        }
        void put_value (int v)
        {
            value = v;
        }
    };
```

Класс *tabl_ident*:

```
    class tabl_ident
    {
        ident    *   p;
        int          size;
        int          top;
    public:
        tabl_ident ( int max_size )
        {
            p    = new ident[size=max_size];
            top  = 1;
        }
        ~tabl_ident ()
        {
            delete []p;
        }
        ident& operator[] ( int k )
        {
```

```
            return p[k];
        }
        int put ( const char *buf );
    };

    int tabl_ident::put ( const char *buf )
    {
        for ( int j=1; j<top; ++j )
            if ( !strcmp(buf, p[j].get_name()) ) return j;
        p[top].put_name(buf);
        ++top;
        return top-1;
    }
```

Класс *Scanner*:

```
    class Scanner
    {
        enum                    state { H, IDENT, NUMB, COM, ALE, DELIM, NEQ };
        static char         *   TW[];
        static type_of_lex      words[];
        static char         *   TD[];
        static type_of_lex      dlms[];
        state                   CS;
        FILE                *   fp;
        char                    c;
        char                    buf[80];
        int                     buf_top;

        void clear ()
        {
            buf_top = 0;
            for ( int j = 0; j < 80; ++j )
                buf[j] = '\0';
        }

        void add ()
        {
            buf [ buf_top ++ ] = c;
        }

        int look ( const char *buf, char **list )
        {
            int i = 0;
            while ( list[i] )
            {
                if ( !strcmp(buf, list[i]) )
                    return i;
                ++i;
            }
            return 0;
        }

        void gc ()
        {
            c = fgetc (fp);
        }
```

```
public:
     Lex get_lex ();

     Scanner ( const char * program )
     {
          fp = fopen ( program, "r" );
          CS = H;
          clear();
          gc();
     }
};
```

Таблицы лексем М-языка можно описать на С++, например, таким образом:

```
char * Scanner::TW[] =
{
     NULL,         // 0
     "and",        // 1
     "begin",      // 2
     "bool",       // 3
     "do",         // 4
     "else",       // 5
     "end",        // 6
     "if",         // 7
     "false",      // 8
     "int",        // 9
     "not",        // 10
     "or",         // 11
     "program",    // 12
     "read",       // 13
     "then",       // 14
     "true",       // 15
     "var",        // 16
     "while",      // 17
     "write",      // 18
     "!",          // 19
     "!F"          // 20
};

char * Scanner:: TD[] =
{
     NULL,         // 0
     ";",          // 1
     "@",          // 2
     ",",          // 3
     ":",          // 4
     ":=",         // 5
     "(",          // 6
     ")",          // 7
     "=",          // 8
     "<",          // 9
     ">",          // 10
     "+",          // 11
     "-",          // 12
     "*",          // 13
     "/",          // 14
     "<=",         // 15
     "!=",         // 16
```

```
        ">="          // 17
    };

    tabl_ident TID(100);

    type_of_lex Scanner::words[] =
    {
        LEX_NULL,
        LEX_AND,
        LEX_BEGIN,
        LEX_BOOL,
        LEX_DO,
        LEX_ELSE,
        LEX_END,
        LEX_IF,
        LEX_FALSE,
        LEX_INT,
        LEX_NOT,
        LEX_OR,
        LEX_PROGRAM,
        LEX_READ,
        LEX_THEN,
        LEX_TRUE,
        LEX_VAR,
        LEX_WHILE,
        LEX_WRITE,
        LEX_NULL
    };

    type_of_lex Scanner::dlms[] =
    {
        LEX_NULL,
        LEX_FIN,
        LEX_SEMICOLON,
        LEX_COMMA,
        LEX_COLON,
        LEX_ASSIGN,
        LEX_LPAREN,
        LEX_RPAREN,
        LEX_EQ,
        LEX_LSS,
        LEX_GTR,
        LEX_PLUS,
        LEX_MINUS,
        LEX_TIMES,
        LEX_SLASH,
        LEX_LEQ,
        LEX_NEQ,
        LEX_GEQ,
        LEX_NULL
    };
```

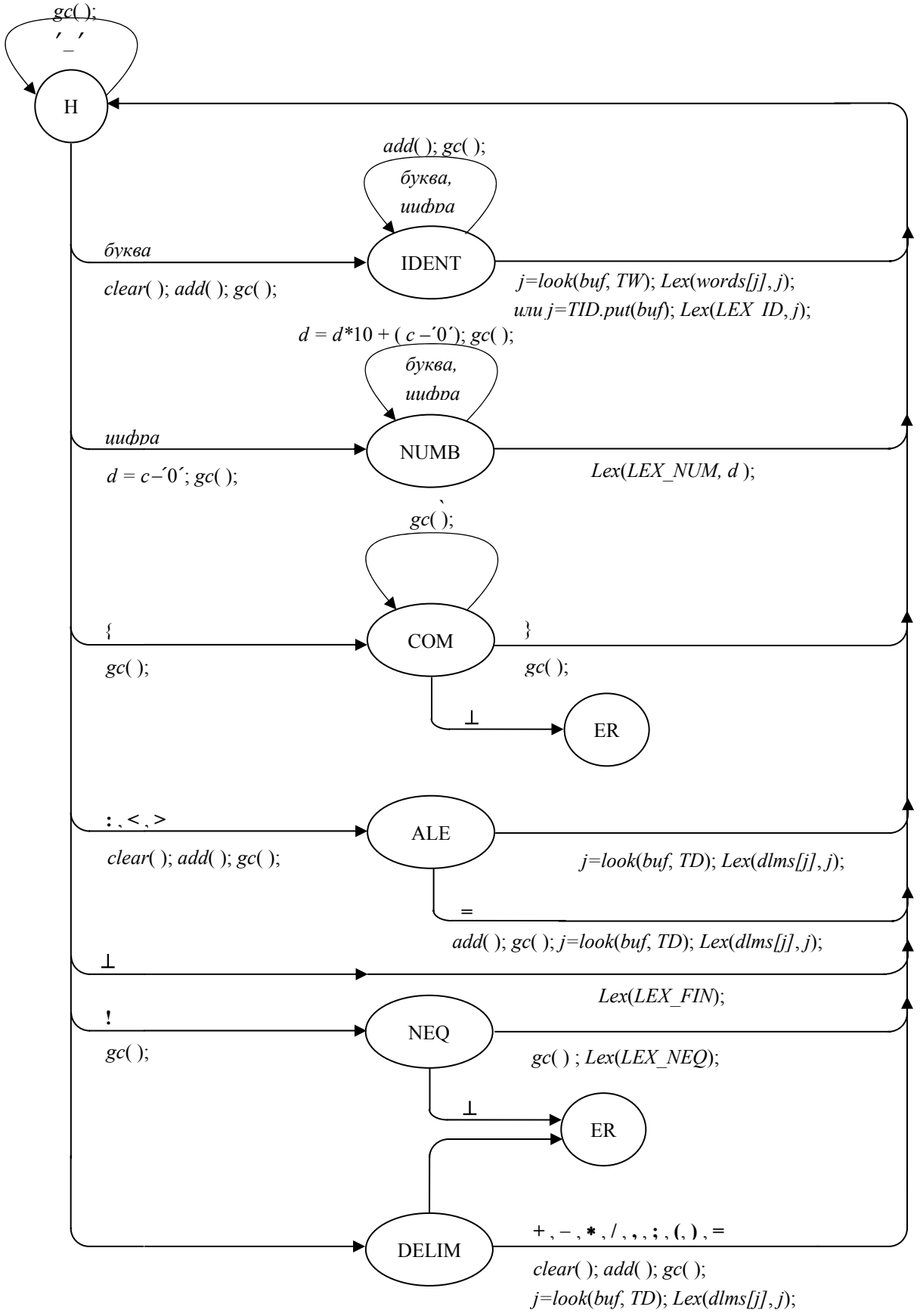


Рис. 7. ДС для модельного языка.

На рисунке 5 приведена диаграмма состояний для модельного 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*);.

Непосредственно реализует **ЛА по ДС** функция *get_lex* ():

```
Lex Scanner::get_lex ()
{
    int d, j;

    CS = H;
    do
    {
        switch ( CS )
        {
        case H:
            if ( c ==' ' || c =='\n' || c=='\r' || c =='\t' )
                gc ();
            else if ( isalpha(c) )
            {
                clear ();
                add ();
                gc ();
                CS = IDENT;
            }
            else if ( isdigit(c) )
            {
                d = c - '0';
                gc ();
                CS = NUMB;
            }
            else if ( c== '{' )
            {
                gc ();
                CS = COM;
            }
            else if ( c== ':' || c== '<' || c== '>')
            {
                clear ();
                add ();
                gc ();
                CS = ALE;
            }
            else if ( c == '@' )
                return Lex(LEX_FIN);
            else if ( c == '!' )
            {
                clear ();
                add ();
                gc ();
                CS = NEQ;
            }
            else
                CS = DELIM;
            break;
        case IDENT:
```

```
            if ( isalpha(c) || isdigit(c) )
            {
                add ();
                gc ();
            }
            else
                if ( j = look (buf, TW) )
                    return Lex (words[j], j);
                else
                {
                    j = TID.put(buf);
                    return Lex (LEX_ID, j);
                }
            break;
        case NUMB:
            if ( isdigit(c) )
            {
                d = d * 10 + (c - '0');
                gc();
            }
            else
                return Lex ( LEX_NUM, d );
            break;
        case COM:
            if ( c == '}' )
            {
                gc ();
                CS = H;
            }
            else if (c == '@' || c == '{' )
                throw c;
            else
                gc ();
            break;
        case ALE:
            if ( c == '=' )
            {
                add ();
                gc ();
                j = look ( buf, TD );
                return Lex ( dlms[j], j );
            }
            else
            {
                j = look (buf, TD);
                return Lex ( dlms[j], j );
            }
            break;
        case NEQ:
            if ( c == '=' )
            {
                add ();
                gc ();
                j = look ( buf, TD );
                return Lex ( LEX_NEQ, j );
            }
            else
                throw '!';
            break;
        case DELIM:
```

```
                    clear ();
                    add ();
                    if (j = look(buf, TD))
                    {
                        gc ();
                        return Lex ( dlms[j], j );
                    }
                    else
                        throw c;
                    break;
                }               // end switch
            }
            while ( true );
        }
```

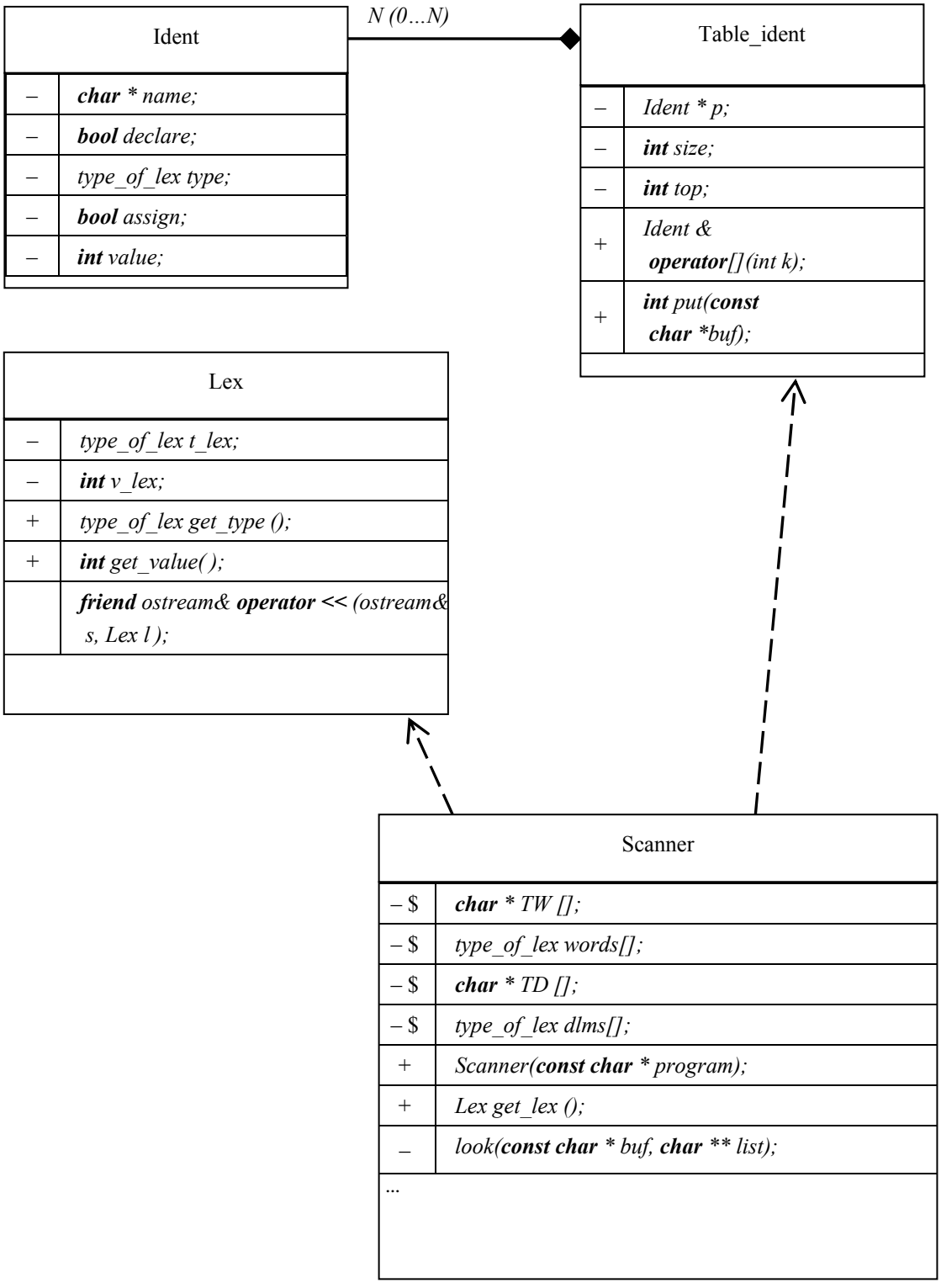


Рис. 8. Иерархия классов для лексического анализатора
(с использованием обозначений диаграмм UML).

Синтаксический анализ

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. Поэтому для описания синтаксиса языков программирования нужны более мощные грамматики, чем регулярные. Обычно для этого 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.

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 ([3]), 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.

Различные методы синтаксического анализа, или разбора, основываются на разных принципах, и используют различные техники построения дерева вывода. Каждый метод предполагает свой способ построения по грамматике программы-анализатора, которая будет осуществлять разбор цепочек. Анализатор некорректен, если:

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

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

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

Метод рекурсивного спуска

Пример: пусть дана грамматика $G_1 = \langle \{a, b, c, d\}, \{S, A, B\}, P, S \rangle$, где

$L(G) = \{c^n abc^m a\ d \mid n, m \geq 0\}$

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 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 (см. рис. 9):

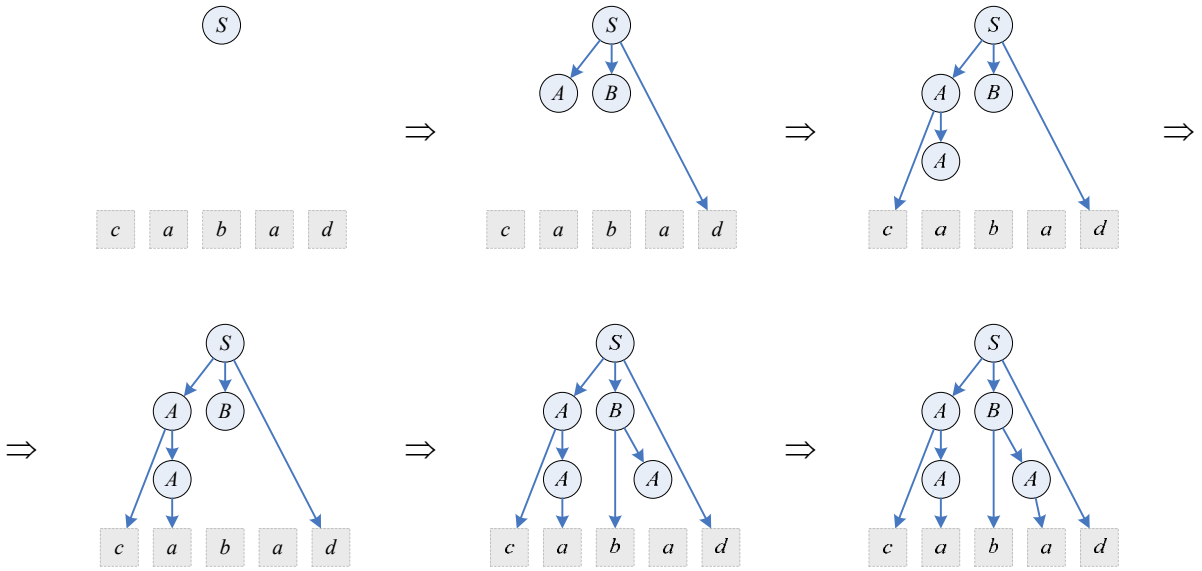


Рис. 9. Построение левого вывода.

Метод рекурсивного 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, что означает, что цепочка не принадлежит языку; разбор останавливается.

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

Тело каждой такой процедуры пишется непосредственно по правилам вывода соответствующего нетерминала: для правой части каждого правила осуществляется поиск подцепочки, выводимой из этой правой части. При этом терминалы распознаются самой процедурой, а нетерминалы соответствуют вызовам процедур, носящих их имена. После распознавания каждого терминала считывается следующий символ из исходной цепочки. Выбор нужной альтернативы осуществляется процедурой по первому символу из еще нерассмотренной части исходной цепочки (т. е. по «текущему» символу).

Работа системы процедур начинается с главной функции `main()`. Она считывает первый символ исходной цепочки и вызывает процедуру `S()`, которая проверяет, выводится ли входная цепочка из S (в общем случае это делается с участием других процедур, которые, в свою очередь рекурсивно могут вызывать и саму `S()` для анализа фрагмента исходной цепочки). Договоримся, что в конце любой анализируемой цепочки всегда присутствует символ $\perp$ (признак конца цепочки[14)], так что в задачу `main()` входит также распознавание этого символа. Можно считать, что `main()`соответствует добавленному в грамматику правилу $M \to S \perp$, где M — новый начальный символ.

Пример: совокупность процедур рекурсивного спуска для грамматики G_1:

$S \to ABd$
$A \to a \mid cA$
$B \to bA$

будет такой:

```
#include <iostream.h>

int c;
void A ();
void B ();
void gc ()
{
    cin >> c;
}

void S ()
{
    cout << "S-->ABd, ";        // применяемое правило вывода
    A();
    B();
    if ( c != 'd' )
        throw c;
}

void A ()
{
    if (c =='a')
    {
        cout << "A-->a, ";
        gc ();
    }
    else if (c =='c')
    {
        cout << "A-->cA, ";
        gc ();
        A ();
    }
    else
        throw c;
}

void B ()
{
```

[14)] Этим признаком в действительности может быть ситуация «конец файла» или «конец строки».

```
        if ( c =='b' )
        {
            cout << "B-->bA, ";
            gc ();
            A ();
        }
        else
            throw c;
    }

    int main ()
    {
        try
        {
            gc ();
            S ();
            if ( c != '⊥' )
                throw c;
            cout << "SUCCESS !!!" << endl;
            return 0;
        }
        catch ( int c )
        {
            cout << "ERROR on lexeme" << c << endl;
            return 1;
        }
    }
```

Для цепочки, выводимой из S, программа напечатает (помимо сообщения об успехе) последовательность правил, применяемых при нисходящем построении дерева вывода для данной цепочки (эта же последовательность годится для построения левого вывода). Вместо печати применяемых правил можно вставить действия по формированию дерева в динамической памяти в виде узлов, связанных указателями. Такое дерево может использоваться на последующих этапах трансляции.

Заметим, что даже если специально не фиксировать структуру анализируемой цепочки, система рекурсивных процедур все равно неявно обходит дерево вывода этой цепочки. Действительно, распознавание терминала a процедурой $B()$ соответствует в дереве вывода ветви $a \overset{B}{\swarrow}$, а вызов процедуры $A()$ из процедуры $B()$ соответствует ветви $B \overset{A}{\searrow}$. Добавив в процедуры анализа дополнительные действия, можно наряду с проверкой синтаксиса вычислять смысл (семантику) входной цепочки. Например, смыслом арифметического выражения является его значение, и оно может быть вычислено в процессе неявного обхода дерева при разборе этого выражения.

Выбор нужной альтернативы при анализе методом рекурсивного спуска можно осуществить, если все альтернативы начинаются попарно различных терминальных символов. Сформулируем **достаточное условие применимости метода рекурсивного спуска**:

каждое правило в грамматике имеет вид

(а) либо $X \to \alpha$,

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

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

где $a_i \in T$ для всех $i = 1, 2, ..., n$; $a_i \neq a_j$ 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 различными;

Это условие не является необходимым.

Метод рекурсивного спуска является разновидностью нисходящего анализа с прогнозируемым выбором альтернатив. Это предполагает, что по грамматике можно заранее предсказать, какую альтернативу нужно будет выбирать на очередном шаге вывода в соответствии с первым символом из непрочитанной части входной цепочки. Далее мы подробно рассмотрим этот подход и сформулируем критерий его применимости.

Нисходящий анализ с прогнозируемым выбором альтернатив

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

$S \to ABd$
$A \to a \mid cA$
$B \to bA$

можно отметить следующее:

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

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

(4) если на каком-то шаге получилась сентенциальная форма вида $w\alpha$, отличная от (2) и (3), где w — максимально длинное начало, состоящее только из терминалов, то если α пуста и w совпадает с анализируемой цепочкой, процесс вывода успешно завершается, иначе фиксируется ошибка: анализируемая цепочка не принадлежит языку $L(G_1)$.

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

	a	b	c	d
S	$S \to ABd$		$S \to ABd$	
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 $wY\alpha$ — очередная сентенциальная форма, где $w \in T^*$. 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 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 $L(G)$.

Не для каждой КС-грамматики существует таблица с однозначными прогнозами, позволяющими сделать правильный выбор альтернативы на каждом шаге вывода. Таким образом, рассмотренный метод пригоден лишь для некоторого подкласса КС-грамматик.

Как мы видели выше, один из способов реализовать программу-анализатор для нисходящего анализа с прогнозируемым выбором альтернатив заключается в построении системы рекурсивных процедур.[15)]

Реализацию нисходящего разбора в виде системы рекурсивных процедур называют *методом рекурсивного спуска*. В нашем курсе под методом рекурсивного спуска подразумевается разбор с помощью анализатора в виде системы рекурсивных процедур, построенного по грамматике на основе таблицы прогнозов. Не для каждой грамматики такая таблица существует, поскольку не всегда возможно заранее спрогнозировать выбор альтернативы: может оказаться, что подходящими в данной ситуации являются сразу несколько альтернатив (неоднозначный прогноз).[16)] Метод рекурсивного спуска *применим* к грамматике, если для нее существует таблица однозначных прогнозов. Это означает, что левый вывод (или дерево нисходящим способом) можно построить, начиная с начального символа S, так, что на каждом шаге вывода решение о том, какое правило (альтернативу) применять для замены левого нетерминала, безошибочно принимается по первому символу из непрочитанной части входной цепочки (т. е. по «текущему» символу). Следовательно, анализатор в виде системы рекурсивных процедур, построенный на основе таблицы прогнозов, будет на любой цепочке работать корректно.

О применимости метода рекурсивного спуска

Очевидно, что метод рекурсивного спуска (без возвратов) неприменим к неоднозначным грамматикам. Например, по грамматике

$$G_2 : S \to aA \mid B \mid d$$
$$A \to d \mid aA$$
$$B \to aA \mid a$$

---

15) Другой способ, с явным использованием стека для хранения нетерминальной части сентенциальной формы, известен как $LL(1)$-анализатор.

16) Обобщение нашего метода — рекурсивный спуск с возвратами — «пробует» по очереди все возможные альтернативы в случае неоднозначного прогноза; о неудаче он сообщает только в том случае, если ни одна из альтернатив не привела к успеху. Если грамматика неоднозначна, обобщенный метод построит все возможные деревья выводов. Мы не рассматриваем подробно рекурсивный спуск с возвратами, поскольку время, затрачиваемое на анализ при таком подходе, может экспоненциально зависеть от длины входной цепочки.

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 $S \to B$).

Как показывает следующий пример, грамматика может быть однозначной, однако однозначных прогнозов для нее также не существует:

$G_3 : S \to A \mid B$
$A \to aA \mid d$
$B \to aB \mid b$

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 $X \to \alpha$ 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*. Так что нисходящий анализ с прогнозируемым выбором альтернатив невозможен по такой грамматике, и метод рекурсивного спуска неприменим. Сформулируем это условие более формально.

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

Например, для альтернатив правила $S \to A \mid B$ в грамматике G_3 имеем: $first(A) = \{ a, d \}$, $first(B) = \{ a, b \}$. Пересечение этих множеств непусто: $first(A) \cap first(B) = \{ a \} \neq \varnothing$, и поэтому метод рекурсивного спуска к G_3 неприменим.

Итак, наличие в грамматике двух различных правил $X \to \alpha \mid \beta$, таких что $first(\alpha) \cap first(\beta) \neq \varnothing$, делает метод рекурсивного спуска неприменимым.

Рассмотрим еще несколько примеров.

$G_4 : S \to aA \mid BDc$
$A \to BAa \mid aB \mid b$
$B \to \varepsilon$
$D \to B \mid b$

$first(aA) = \{ a \}, first(BDc) = \{ b, c \};$
$first(BAa) = \{ a, b \}, first(aB) = \{ a \}, first(b) = \{ b \};$
$first(\varepsilon) = \varnothing ;$
$first(B) = \varnothing, first(b) = \{ b \}.$

Метод рекурсивного спуска неприменим к грамматике G_4, так как $first(BAa) \cap first(aB) = \{ a \} \neq \varnothing$.

Заметим, что для любой грамматики справедливо следующее: если $\alpha \Rightarrow \varepsilon$, то $first(\alpha) = \varnothing$. Ниже будет показано, что прогнозирование с альтернативами, из которых выводится пустая цепочка, требует, помимо *first*, проверки еще некоторых свойств грамматики.

$G_5 : S \to aA$
$A \to BC \mid B$
$C \to b \mid \varepsilon$
$B \to \varepsilon$

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 спус-

ка к ней неприменим. Действительно, $BC \Rightarrow \varepsilon$ и $B \Rightarrow \varepsilon$. Цепочка a имеет два различных дерева вывода (см. рис. 10):

Рис. 10. Два различных дерева выводва для цепочки a.

Таким образом, если в грамматике для двух различных правил $X \to \alpha \mid \beta$ 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.

G_6 :
$S \to cAd \mid d$
$A \to aA \mid \varepsilon$
$\left| \; first(cAd) = \{ c \}, first(d) = \{ d \}; \right.$

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

Выбор альтернативы для A 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. Таблица прогнозов для G_6:

	a	c	d
S		$S \to cAd$	$S \to d$
A	$A \to a\,A$	$A \to \varepsilon$	$A \to \varepsilon$

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

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

G_7:
$S \to Bd$
$B \to cAa \mid a$
$A \to aA \mid \varepsilon$
$\left| \; first(cAa) = \{ c \}, first(a) = \{ a \}; \right.$

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

Однако для нетерминала A прогноз по символу a неоднозначен. Дело в том, что любой вывод, содержащий A, имеет вид: $S \to Bd \to cAad \to \ldots \to ca\ldots aAad$, и поэтому альтернативу $A \to \varepsilon$ следует применять только в том случае, если текущий символ a, а следующий за ним отличен от a (например, d). Если следующий за текущим символом — тоже символ a, то выбирается альтернатива $A \to aA$. Но сделать однозначный выбор по текущему символу в пользу какой-то одной из этих альтернатив невозможно, т. к. 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 пустую альтернативу, стоит символ a, с которого также начинается и непустая альтернатива для A. В таком случае процедура $A()$ не сможет правильно определить, 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.

Определение: *множество follow(A)* — это множество терминальных символов, которые следуют за цепочками, выводимыми из A в грамматике $G = \langle T, N, P, S \rangle$, 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 пара правил $X \to \alpha \mid \beta$, таких что $\beta \Rightarrow \varepsilon$, $first(\alpha) \cap follow(X) \neq \varnothing$, то метод рекурсивного спуска неприменим к данной грамматике.**

Итак, на примерах мы рассмотрели все случаи, когда можно построить однозначные прогнозы по грамматике. Подытожив их, сформулируем **критерий применимости** метода рекурсивного спуска.

Утверждение 11. Пусть G — КС-грамматика. Метод рекурсивного спуска применим к G если и только и если для любых ее двух правил $X \to \alpha \mid \beta$ 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 $first(\alpha) \cap follow(X) = \varnothing$.

Рассмотрим грамматику

G_8:

$S \to BDC$ $C \to Bd$ $D \to aB \mid d$ $B \to bB \mid \varepsilon$	$first\,(aB) = \{ a \}$, $first\,(d) = \{ d \}$; $first\,(bB) = \{ b \}$, $follow\,(B) = \{a,b,d\}$ так как следующие возможны сентенциальные формы: $B\mathbf{d}C$, $B\mathbf{a}B\mathbf{b}d$ [17].

Поскольку $first\,(bB) \cap follow\,(B) = \{ b \} \neq \varnothing$, метод рекурсивного спуска неприменим к данной грамматике.

Естественно задаться вопросом: если грамматика не удовлетворяет критерию применимости метода рекурсивного спуска, то есть ли возможность построить эквивалентную грамматику, к которой данный метод применим.

Утверждение 12. Не существует алгоритма, определяющего для произвольной КС-грамматики, существует ли для нее эквивалентная грамматика, к которой метод рекурсивного спуска применим (т. е. это алгоритмически неразрешимая проблема[18]).

Построение таблицы прогнозов

Если критерий применимости метода рекурсивного спуска выполняется для грамматики G, то таблицу M однозначных прогнозов можно построить следующим образом:

1. Для каждого правила $X \to \alpha$ и для каждого терминала $a \in first(\alpha)$ помещаем $X \to \alpha$ в ячейку $M\,[X, a]$;

2. Для каждого правила $X \to \alpha$, такого что $\alpha \Rightarrow \varepsilon$, помещаем $X \to \alpha$ во все незаполненные на 1-м шаге ячейки строки X.

Рекурсивный спуск без использования прогнозов

Выделим подкласс грамматик, по которым можно строить систему рекурсивных процедур, минуя построение таблицы прогнозов.

Будем говорить, что КС-грамматика имеет **канонический вид для метода рекурсивного спуска** (т. е. в форму, удобную для построения рекурсивных процедур), если каждая группа правил с одинаковой левой частью имеет один из перечисленных ниже видов и выполняются дополнительные условия:

(а) либо $X \to \alpha$,

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

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

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 долж-

---

[17] В наших примерах мы вычисляем *first* и *follow* «интуитивно», опираясь на определения. Алгоритмы вычисления множеств *first* и *follow* читателю предлагается сформулировать самостоятельно или найти их в литературе, например в [3].

[18] Напомним, что алгоритмическая неразрешимость проблемы означает не то, что данную задачу нельзя решить для каждой конкретной грамматики, а то, что нет универсального способа решения, пригодного для любой грамматики.

ны начинаться с терминалов, причем все эти терминалы должны быть различными;

(в) либо $X \to a_1\alpha_1 \mid a_2\alpha_2 \mid ... \mid a_n\alpha_n \mid \varepsilon$,

где $a_i \in T$ для всех $i = 1, 2,..., n$; $a_i \neq a_j$ для $i \neq j$; $\alpha_i \in (T \cup N)^*$, и $first(X) \cap follow(X) = \varnothing$.

Ясно, что если правила вывода имеют такой вид, то рекурсивный спуск может быть реализован по выше изложенной схеме[19)].

Итераторы в КС-грамматиках

При описании синтаксиса языков программирования часто встречаются правила, описывающие последовательность однотипных конструкций, отделенных друг от друга каким-либо знаком-разделителем (например, список идентификаторов при описании переменных, список параметров при вызове процедур и функций и т. п.). Такие правила обычно имеют вид: $L \to a \mid a, L$

Вместо обычных КС-грамматик для описания синтаксиса языков программирования нередко используют их модификацию, так называемую БНФ[20)], в которой, в частности, допускаются конструкции вида «{α}», где фигурные скобки — это спецсимволы для выделения фрагмента α , который может отсутствовать или повторяться произвольное число раз. Называют такую конструкцию *итератором*.

С помощью итератора грамматику $L \to a \mid a, L$ можно переписать так: $L \to a \ \{, a\}$.

Наоборот, если в грамматике есть правило вида $X \to \alpha\{\beta\}\gamma$, содержащее итератор $\{\beta\}$, то его можно заменить серией эквивалентных правил без итератора $\{\beta\}$:

$X \to \alpha Y \gamma$

$Y \to \beta Y \mid \varepsilon$,

где Y — новый нетерминальный символ, добавляемый в алфавит нетерминалов грамматики.

Чтобы применить метод рекурсивного спуска для грамматики $L \to a \ \{, a\}$, преобразуем эту грамматику в эквивалентную без итератора:

$L \to a\ M$

$M \to , a \mid \varepsilon$

Метод применим к данной грамматике, так как $first(\ , a\ ) \cap follow(M) = \varnothing$.

Можно построить систему рекурсивных процедур по преобразованной грамматике, но лучше, убедившись, что для преобразованной грамматики метод применим, вернуться к начальному варианту с итератором.

Реализовать итератор $\{\beta\}$ (где $\beta = b\beta'$, $b \in T$) удобно с помощью цикла: «пока текущий символ равен b, считать следующий символ и проанализировать цепочку β′».

Запишем такую реализацию для грамматики $L \to$ a {, a}.

```
void L ()
{
    if ( c != 'a' )
        throw c;
```

[19)] Канонический вид дает достаточное, но не необходимое условие применимости метода рекурсивного спуска.

[20)] Бэкуса-Наура форма.

```
        gc ();
        while ( c == ',' )
        {
            gc ();
            if ( c != 'a' )
                throw c;
            else
                gc ();
        }
    }
```

Рассмотрим пример еще одной грамматики.

$G_{sequence}$: $S \to LB\perp$

$L \to a\ \{, a\}$

$B \to , b$

Если для этой грамматики сразу написать анализатор, не убедившись в применимости метода к преобразованной (без итератора) грамматике, то цепочка *a*,*a*,*a*,*b* будет признана таким анализатором ошибочной, хотя в действительности *a*,*a*,*a*,*b* принадлежит языку $G_{sequence}$. Это произойдет потому, что процедура *L*() захватит «чужую» запятую — ту, что выводится из *B*, и далее не обнаружив символ *a*, сообщит об ошибке.

Если же вначале проверить применимость метода, исключив из грамматики итератор рассмотренным выше способом:

$S \to LB\perp$

$L \to a\ M$

$M \to , a \mid \varepsilon$

$B \to , b$

то нетрудно видеть, что $first\ (, a) \cap follow\ (M) = \{ , \} \neq \varnothing$ и поэтому метод рекурсивного спуска неприме́ним[21]. Можно попытаться поискать другую эквивалентную грамматику, к которой метод приме́ним. Некоторые полезные эквивалентные преобразования грамматик будут рассмотрены ниже. Однако, как следует из утверждения 12, успех в поиске эквивалентной грамматики, для которой метод приме́ним, с помощью каких-либо эквивалентных преобразований, не гарантирован.

Преобразования грамматик

Если грамматика не удовлетворяет требованиям применимости метода рекурсивного спуска, то можно попытаться **преобразовать** ее, т. е. получить эквивалентную грамматику, пригодную для анализа этим методом.

1) Если в грамматике есть нетерминалы, правила вывода которых **непосредственно леворекурсивны**, т. е. имеют вид

$A \to A\alpha_1 \mid ... \mid A\alpha_n \mid \beta_1 \mid ... \mid \beta_m$,

где $\alpha_i \in (T \cup N)^+$ для $i = 1, 2, ..., n$; $\beta_j \in (T \cup N)^*$ для $j = 1, 2, ..., m$,

[21] Если бы в $G_{sequence}$ последовательность терминалов, перечисляемых через запятую завершалась бы отличным от запятой символом (например, точкой с запятой), как это обычно и бывает в реальных языках программирования, то метод рекурсивного спуска был бы приме́ним.

то в таком случае применять метод рекурсивного спуска нельзя, поскольку $first(A\alpha_i) \cap first(A\alpha_k) \neq \varnothing$ для некоторых $i \neq k$, или $\beta_j = \varepsilon$ для некоторого j и $first(A\alpha_i) \cap follow(A) \neq \varnothing$ для $i = 1, 2, ..., n$.

Левую рекурсию всегда можно заменить правой:

$A \to \beta_1 A' \mid ... \mid \beta_m A'$

$A' \to \alpha_1 A' \mid ... \mid \alpha_n A' \mid \varepsilon$

Будет получена грамматика, эквивалентная данной, т.к. из нетерминала A по-прежнему выводятся цепочки вида $\beta_j \{\alpha_i\}$, где $i = 1, 2, ... ,n; j = 1, 2, ... , m$.

2) Если в грамматике есть нетерминал, у которого несколько правил вывода начинаются **одинаковыми терминальными символами**, т. е. имеют вид

$A \to a\alpha_1 \mid a\alpha_2 \mid ... \mid a\alpha_n \mid \beta_1 \mid ... \mid \beta_m$,

где $a \in T$; $\alpha_i, \beta_j \in (T \cup N)^*$, β_j не начинается с a, $i = 1, 2, ... , n, j = 1, 2, ... , m$,

то непосредственно применять метод рекурсивного спуска нельзя, т. к. $first(a\alpha_i) \cap first(a\alpha_k) \neq \varnothing$ для $i \neq k$. Можно преобразовать правила вывода данного нетерминала, объединив правила с общими началами в одно правило:

$A \to aA' \mid \beta_1 \mid ... \mid \beta_m$

$A' \to \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$

Будет получена грамматика, эквивалентная данной.

3) Если в грамматике есть нетерминал, у которого **несколько** правил вывода, и среди них есть правила, **начинающиеся нетерминальными символами**, т. е. имеют вид:

$A \to B_1\alpha_1 \mid ... \mid B_n\alpha_n \mid a_1\beta_1 \mid ... \mid a_m\beta_m$

$B_1 \to \gamma_{11} \mid ... \mid \gamma_{1k}$

. . .

$B_n \to \gamma_{n1} \mid ... \mid \gamma_{np}$, где $B_i \in N$; $a_j \in T$; $\alpha_i, \beta_j, \gamma_{ij} \in (T \cup N)^*$,

то можно заменить вхождения нетерминалов B_i их правилами вывода в надежде, что правила нетерминала A станут удовлетворять условиям применимости метода рекурсивного спуска:

$A \to \gamma_{11}\alpha_1 \mid ... \mid \gamma_{1k}\alpha_1 \mid ... \mid \gamma_{n1}\alpha_n \mid ... \mid \gamma_{np}\alpha_n \mid a_1\beta_1 \mid ... \mid a_m\beta_m$

4) Если есть правила с пустой альтернативой вида:

$A \to \alpha_1 A \mid ... \mid \alpha_n A \mid \beta_1 \mid ... \mid \beta_m \mid \varepsilon$

$B \to \alpha A \beta$

и $first(A) \cap follow(A) \neq \varnothing$ (из-за вхождения A в правила вывода для B), то можно попытаться преобразовать такую грамматику:

$B \to \alpha A'$

$A' \to \alpha_1 A' \mid ... \mid \alpha_n A' \mid \beta_1\beta \mid ... \mid \beta_m\beta \mid \beta$

Полученная грамматика будет эквивалентна исходной, т. к. из B по-прежнему выводятся цепочки вида $\alpha \{\alpha_i\} \beta_j \beta$ либо $\alpha \{\alpha_i\} \beta$.

Однако правило вывода для нетерминального символа A' будет иметь альтернативы, начинающиеся одинаковыми терминальными символами (т. к. $first(A) \cap follow(A) \neq \varnothing$, следовательно, потребуются дальнейшие преобразования, и успех не гарантирован.

Пример. Рассмотрим грамматику G_{origin}:

$S \to fASd \mid \varepsilon$

$A \to Aa \mid Ab \mid dB \mid f$

$B \to bcB \mid \varepsilon$

$first(Aa) = first(Ab) = \{d, f\}$

$first(bcB) = \{b\}, follow(B) = \{a, b, d, f\}$

Условия применимости метода рекурсивного спуска не выполняются для G_{origin}. С помощью преобразований приведем эту грамматику к каноническому виду для рекурсивного спуска.

G_{origin}:

$S \to fASd \mid \varepsilon$

$A \to \underline{Aa} \mid \underline{Ab} \mid dB \mid f$

$B \to bcB \mid \varepsilon$

$\Rightarrow$

$G_{transform_1}$:

$S \to fASd \mid \varepsilon$

$A \to dBA' \mid fA'$

$A' \to aA' \mid bA' \mid \varepsilon$

$B \to bcB \mid \varepsilon$

$first(S) = \{f\}, follow(S) = \{d\};$

$first(A') = \{a, b\}, follow(A') = \{f, d\};$

$first(B) = \{b\}, follow(B) = \{a, b, f, d\};$

$first(S) \cap follow(S) = \varnothing$; $first(A') \cap follow(A') = \varnothing;$

$first(B) \cap follow(B) = \{b\} \neq \varnothing;$

$G_{transform_2}$:

$\Rightarrow$

$S \to fASd \mid \varepsilon$

$A \to dB' \mid fA'$

$B' \to bcB' \mid \underline{A'}$

$A' \to aA' \mid bA' \mid \varepsilon$

~~$B \to bcB \mid \varepsilon$~~

недостижимые правила, их можно убрать

$G_{transform_3}$:

$\Rightarrow$

$S \to fASd \mid \varepsilon$

$A \to dB' \mid fA'$

$B' \to \underline{bcB'} \mid aA' \mid \underline{bA'} \mid \varepsilon$

$A' \to aA' \mid bA' \mid \varepsilon$

$G_{transform_4}$:

$\Rightarrow$

$S \to fASd \mid \varepsilon$

$A \to dB' \mid fA'$

$B' \to bC \mid aA' \mid \varepsilon$

$C \to cB' \mid \underline{A'}$

$A' \to aA' \mid bA' \mid \varepsilon$

G_{object}:

$\Rightarrow$

$S \to fASd \mid \varepsilon$

$A \to dB' \mid fA'$

$B' \to bC \mid aA' \mid \varepsilon$

$C \to cB' \mid aA' \mid bA' \mid \varepsilon$

$A' \to aA' \mid bA' \mid \varepsilon$

$first(B') = \{a, b\}, follow(B') = \{f, d\}; first(B') \cap follow(B') = \varnothing;$

$first(A') = \{a, b\}, follow(A') = \{f, d\}; first(A') \cap follow(A') = \varnothing;$

$first(C) = \{a, b, c\}, follow(C) = \{f, d\}; first(C) \cap follow(C) = \varnothing.$

Т. е. получили эквивалентную грамматику G_{object}, к которой применим метод рекурсивного спуска.

Задача разбора для неоднозначных грамматик

Для неоднозначных грамматик задача синтаксического анализа (задача разбора) может быть поставлена двумя основными способами.

(1) *Даны КС-грамматика G и цепочка x. Требуется проверить: $x \in L(G)$? Если да, то построить все деревья вывода для x (или все левые выводы для x, или все правые выводы для x)*[22)].

Для решения этой задачи можно обобщить метод рекурсивного спуска, чтобы он работал с возвратами, пробуя различные подходящие альтернативы.

(2) *Даны КС-грамматика G и цепочка x. Требуется проверить: $x \in L(G)$? Если да, то построить одно дерево вывода для x (возможно, «наиболее подходящее» в некотором смысле).*

Если требуется построить только одно («наиболее подходящее») дерево вывода цепочки в неоднозначной грамматике, то для некоторых грамматик анализ рекурсивным спуском (без возвратов) можно осуществить так: при неоднозначном прогнозе предпочтение в выборе всегда отдаем какой-то одной «наиболее подходящей» альтернативе. Для некоторых неоднозначных грамматик построенный таким образом анализатор работает корректно, и строит «наиболее подходящее» в некотором смысле дерево.

Рассмотрим пример, иллюстрирующий ситуацию с условными (полным и сокращенным) операторами в языке Паскаль.

$$G_{\text{if-then}} = \langle \{\textbf{\textit{if}}, \textbf{\textit{then}}, \textbf{\textit{else}}, a, b\}, \{S\}, P, S, S' \rangle,$$

где $P = \{ S \to \textbf{if}\ b\ \textbf{then}\ S\ S' \mid a;\ S' \to \textbf{\textit{else}}\ S \mid \varepsilon \}$. В этой грамматике прогноз для S' по ***else*** неоднозначен, так как $first(\textbf{else}\ S) \cap follow(S') = \{\textbf{\textit{else}}\} \neq \varnothing$. Для цепочки ***if*** *b* ***then if*** *b* ***then*** *a* ***else*** *a* можно построить два различных дерева вывода, показанных на рис. 11 (а, б).

Если при построении анализатора отдать предпочтение непустой альтернативе для S', то такой анализатор построит дерево, изображенное на рис. 11 (а), в котором ***else*** соотносится с ближайшим (на его уровне вложенности) ***if***, что соответствует правилам, принятым в языке Паскаль при разрешении подобных неоднозначностей в комбинациях условных операторов.

Нетрудно видеть, что для данного примера неоднозначной грамматики анализатор, построенный с предпочтением выбора непустой альтернативы для нетерминала S', корректен. Заметим, что мы модифицировали метод рекурсивного спуска для данного примера, отдав предпочтение одной из альтернатив. Анализ рекурсивным спуском в «чистом виде» для грамматики $G_{if\text{-}then}$ невозможен, так как она неоднозначна и поэтому не удовлетворяет критерию применимости.

[22)] Цепочка в неоднозначной грамматике может иметь и бесконечно много деревьев вывода. В таком случае можно ограничиться построением всех деревьев, высота которых не превосходит некоторой константы.

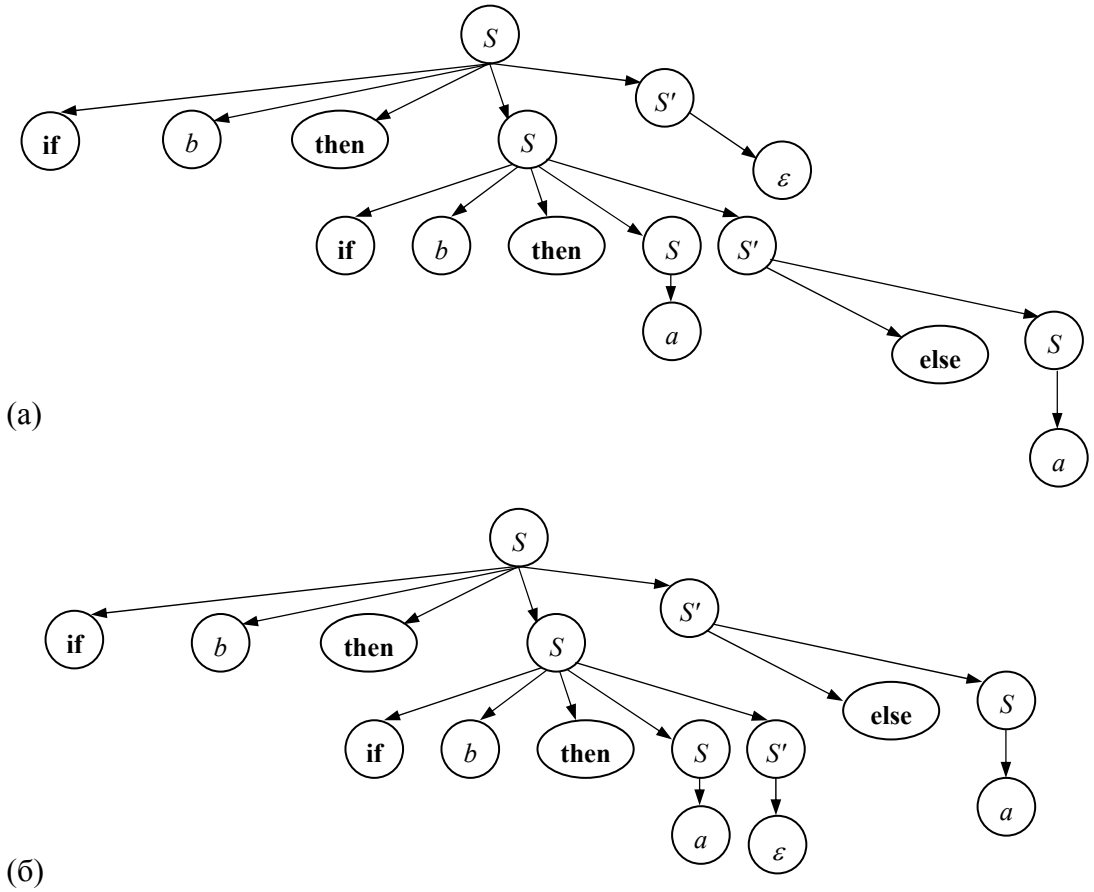


Рис. 11. Деревья вывода для цепочки if *b* then if *b* then *a* else *a*.

О других методах распознавания КС-языков

Метод рекурсивного спуска применим к достаточно узкому подклассу КС-грамматик. Известны более широкие подклассы КС-грамматик, для которых существуют эффективные анализаторы, обладающие тем же свойством, что и анализатор, написанный методом рекурсивного спуска, — входная цепочка считывается один раз слева направо и процесс разбора полностью детерминирован, в результате на обработку цепочки длины n расходуется время cn. К таким грамматикам 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 (см., например, [2], [3]).

Анализатор (распознаватель) 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.

Анализатор (распознаватель) 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 «задом наперед», как при разборе по леволинейной автоматной грамматике. По $LR(k)$-грамматике строится таблица, которая на каждом шаге вывода позволяет анализатору однозначно выбрать нужную свертку.

Анализатор (распознаватель) для грамматик предшествования просматривает входную цепочку слева направо и осуществляет детерминированный правый вывод,

учитывая только некоторые отношения между парами смежных символов выводимой цепочки.

Часто одна и та же КС-грамматика может быть отнесена не к одному, а сразу к нескольким классам грамматик (например, любая *LL*-грамматика является *LR*-грамматикой — обратное неверно), допускающих построение линейных распознавателей. Но, на вопрос, какой именно распознаватель выбрать, нисходящий или восходящий эффективнее, нет однозначного ответа.

Нисходящий синтаксический анализ предпочтителен с точки зрения процесса трансляции, поскольку на его основе легче организовать процесс порождения цепочек результирующего языка. Восходящий синтаксический анализ привлекательнее тем, что часто для данного языка программирования легче построить LR-грамматику, а на ее основе — правосторонний распознаватель.

Конкретный выбор зависит от конкретного компилятора, от сложности грамматики входного языка программирования и от того, как будут использованы результаты работы распознавателя.

Синтаксический анализатор для М-языка

Будем считать, что синтаксический и лексический анализаторы взаимодействуют следующим образом: анализ исходной программы идет под управлением синтаксического анализатора; если для продолжения анализа ему нужна очередная лексема, то он запрашивает ее у лексического анализатора; тот выдает одну лексему и «замирает» до тех пор, пока синтаксический анализатор не запросит следующую лексему.

Соглашения:

- наш лексический анализатор — это функция-член класса *Scanner – Lex get_lex* () —, которая в качестве результата выдает лексемы типа (class) *Lex*;
- в переменной *Lex curr_lex* будем хранить текущую лексему, выданную лексическим анализатором, а в переменной *c_val* — ее значение.

Тогда метод рекурсивного спуска можно реализовать на C++, например, следующим образом:

```
class Parser
{
    Lex          curr_lex;
    type_of_lex  c_type;
    int          c_val;
    Scanner      scan;
    Stack < int, 100 > st_int;
    Stack < type_of_lex, 100 > st_lex;

    void P();
    void D1();
    void D();
    void B();
    void S();
    void E();
    void E1();
    void T();
    void F();
    void dec ( type_of_lex type);
    void check_id ();
    void check_op ();
```

```
        void check_not ();
        void eq_type ();
        void eq_bool ();
        void check_id_in_read ();

        void gl ()
        {
            curr_lex = scan.get_lex();
            c_type = curr_lex.get_type();
            c_val = curr_lex.get_value();
        }

    public:
        Poliz    prog;
        Parser ( const char *program) : scan (program), prog (1000) {}
        void     analyze();
    };

    void Parser::analyze ()
    {
        gl ();
        P ();
        prog.print();
        cout << endl << "Yes!!!" << endl;
    }

    void Parser::P ()
    {
        if ( c_type == LEX_PROGRAM )
            gl ();
        else
            throw curr_lex;
        D1 ();
        if ( c_type == LEX_SEMICOLON )
            gl ();
        else
            throw curr_lex;
        B ();
        if ( c_type != LEX_FIN )
            throw curr_lex;
    }

    void Parser::D1 ()
    {
        if ( c_type == LEX_VAR )
        {
            gl ();
            D ();
            while ( c_type == LEX_COMMA )
            {
                gl();
                D();
            }
        }
        else
            throw curr_lex;
    }

    void Parser::D ()
    {
```

```
            st_int.reset();
            if (c_type != LEX_ID)
                throw curr_lex;
            else {
                st_int.push ( c_val );
                gl ();
                while ( c_type == LEX_COMMA )
                {
                    gl();
                    if (c_type != LEX_ID)
                        throw curr_lex;
                    else {
                        st_int.push ( c_val ); gl();
                    }
                }
                if ( c_type != LEX_COLON )
                    throw curr_lex;
                else
                {
                    gl ();
                    if ( c_type == LEX_INT )
                    {
                        dec ( LEX_INT );
                        gl();
                    }
                    else
                    if ( c_type == LEX_BOOL )
                    {
                        dec ( LEX_BOOL );
                        gl();
                    }
                    else
                        throw curr_lex;
                }
            }
        }

        void Parser::B ()
        {
            if (c_type == LEX_BEGIN)
            {
                gl();
                S();
                while (c_type == LEX_SEMICOLON)
                {
                    gl();
                    S();
                }
                if ( c_type == LEX_END )
                    gl();
                else
                    throw curr_lex;
            }
            else
                throw curr_lex;
        }

        void Parser::S ()
        {
            int pl0, pl1, pl2, pl3;
```

```
        if (c_type == LEX_IF)
        {
            gl();
            E();
            eq_bool();
            pl2 = prog.get_free ();
            prog.blank();
            prog.put_lex (Lex(POLIZ_FGO));
            if (c_type == LEX_THEN)
            {
                gl();
                S();
                pl3 = prog.get_free();
                prog.blank();
                prog.put_lex (Lex(POLIZ_GO));
                prog.put_lex (Lex(POLIZ_LABEL, prog.get_free()),pl2);
                if (c_type == LEX_ELSE)
                {
                    gl();
                    S();
                    prog.put_lex(Lex(POLIZ_LABEL,prog.get_free()),pl3);
                }
                else
                    throw curr_lex;
            }
            else
                throw curr_lex;
        } //end if
        else
        if (c_type == LEX_WHILE)
        {
            pl0 = prog.get_free();
            gl();
            E();
            eq_bool();
            pl1 = prog.get_free();
            prog.blank();
            prog.put_lex (Lex(POLIZ_FGO));
            if (c_type == LEX_DO)
            {
                gl();
                S();
                prog.put_lex (Lex (POLIZ_LABEL, pl0));
                prog.put_lex (Lex ( POLIZ_GO));
                prog.put_lex (Lex(POLIZ_LABEL, prog.get_free()),pl1);
            }
            else
                throw curr_lex;
        } //end while
        else
        if ( c_type == LEX_READ )
        {
            gl();
            if (c_type == LEX_LPAREN)
            {
                gl();
                if (c_type == LEX_ID)
                {
                    check_id_in_read();
```

```
                    prog.put_lex (Lex ( POLIZ_ADDRESS, c_val) );
                    gl();
                }
                else
                    throw curr_lex;
                if ( c_type == LEX_RPAREN )
                {
                    gl();
                    prog.put_lex (Lex (LEX_READ));
                }
                else
                    throw curr_lex;
            }
            else
                throw curr_lex;
        } //end read
        else
            if (c_type == LEX_WRITE)
            {
                gl();
                if (c_type == LEX_LPAREN)
                {
                    gl();
                    E();
                    if (c_type == LEX_RPAREN)
                    {
                      gl();
                      prog.put_lex (Lex(LEX_WRITE));
                    }
                    else
                      throw curr_lex;
                }
                else
                    throw curr_lex;
            } //end write
        else
            if ( c_type == LEX_ID )
            {
                check_id ();
                prog.put_lex (Lex(POLIZ_ADDRESS,c_val));
                gl();
                if ( c_type == LEX_ASSIGN )
                {
                    gl();
                    E(); eq_type();
                    prog.put_lex (Lex (LEX_ASSIGN) );
                }
                else
                    throw curr_lex;
            } //assign-end
        else B();
    }

    void Parser::E ()
    {
        E1();
        if ( c_type == LEX_EQ || c_type == LEX_LSS || c_type == LEX_GTR ||
             c_type == LEX_LEQ || c_type == LEX_GEQ || c_type == LEX_NEQ )
        {
            st_lex.push (c_type);
```

```
            gl();
            E1();
            check_op();
        }
    }

    void Parser::E1 ()
    {
        T();
        while ( c_type==LEX_PLUS || c_type==LEX_MINUS || c_type==LEX_OR )
        {
            st_lex.push (c_type);
            gl();
            T();
            check_op();
        }
    }

    void Parser::T ()
    {
        F();
        while ( c_type==LEX_TIMES || c_type==LEX_SLASH || c_type==LEX_AND )
        {
            st_lex.push (c_type);
            gl();
            F();
            check_op();
        }
    }

    void Parser::F ()
    {
        if ( c_type == LEX_ID )
        {
            check_id();
            prog.put_lex (Lex (LEX_ID, c_val));
            gl();
        }
        else
        if ( c_type == LEX_NUM )
        {
            st_lex.push ( LEX_INT );
            prog.put_lex ( curr_lex );
            gl();
        }
        else
        if ( c_type == LEX_TRUE )
        {
            st_lex.push ( LEX_BOOL );
            prog.put_lex (Lex (LEX_TRUE, 1) );
            gl();
        }
        else
        if ( c_type == LEX_FALSE)
        {
            st_lex.push ( LEX_BOOL );
            prog.put_lex (Lex (LEX_FALSE, 0) );
            gl();
        }
        else
```

```
        if (c_type == LEX_NOT)
        {
            gl();
            F();
            check_not();
        }
        else
        if ( c_type == LEX_LPAREN )
        {
            gl();
            E();
            if ( c_type == LEX_RPAREN)
                gl();
            else
                throw curr_lex;
        }
        else
            throw curr_lex;
    }
```

Семантический анализатор для М-языка

Контекстные условия, выполнение которых нам надо контролировать в программах на М-языке, таковы:

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 (как в Паскале).

Проверку контекстных условий совместим с синтаксическим анализом. Для этого в синтаксические правила вставим вызовы процедур, осуществляющих необходимый контроль, а затем перенесем их в процедуры рекурсивного спуска.

Для реализации семантического анализа будем использовать следующий шаблонный класс *Stack*:

```
    template <class T, int max_size > class Stack
    {
        T s[max_size];
        int top;
    public:
        Stack(){top = 0;}
        void reset() { top = 0; }
        void push(T i);
        T pop();
        bool is_empty(){ return top == 0; }
        bool is_full() { return top == max_size; }
    };

    template <class T, int max_size >
```

```
void Stack <T, max_size >::push(T i)
{
    if ( !is_full() ) { s[top] = i; top++; }
    else throw "Stack_is_full";
}

template <class T, int max_size >

T Stack <T, max_size >::pop()
{
    if ( !is_empty() )
    {
        --top;
        return s[top];
    }
    else
        throw "Stack_is_empty";
}
```

Обработка описаний

Для контроля согласованности типов в выражениях и типов выражений в операторах, необходимо знать типы переменных, входящих в эти выражения. Кроме того, нужно проверять, нет ли повторных описаний идентификаторов. Эта информация становится известной в тот момент, когда синтаксический анализатор обрабатывает описания. Следовательно, в синтаксические правила для описаний нужно вставить действия, с помощью которых будем запоминать типы переменных и контролировать единственность их описания.

Лексический анализатор запомнил в таблице идентификаторов *TID* все идентификаторы-лексемы, которые были им обнаружены в тексте исходной программы. Информация о типе переменных и о наличии их описания заносится в ту же таблицу.

I-ая строка таблицы *TID* соответствует идентификатору-лексеме вида (*LEX_ID*, *i*).

Лексический анализатор заполнил поле *name*; значения полей *declare* и *type* будем заполнять на этапе семантического анализа.

Раздел описаний имеет вид

$D \rightarrow I\ \{, I\}$: [***int*** | ***bool***],

т. е. имени типа (***int*** или ***bool***) предшествует список идентификаторов. Эти идентификаторы (вернее, номера соответствующих им строк таблицы TID) надо запоминать (мы будем их запоминать в стеке целых чисел *Stack<int,100> st_int*), а когда будет проанализировано имя типа, надо заполнить поля *declare* и *type* в этих строках.

Функция ***void*** *Parser::dec* (*type_of_lex type*) считывает из стека номера строк таблицы *TID*, заносит в них информацию о типе соответствующих переменных, о наличии их описаний и контролирует повторное описание переменных.

```
void Parser::dec ( type_of_lex type )
{
    int i;
    while ( !st_int.is_empty())
    {
        i = st_int.pop();
        if ( TID[i].get_declare() )
```

```
                throw "twice";
            else
            {
                TID[i].put_declare();
                TID[i].put_type(type);
            }
        }
    }
```

С учетом имеющихся функций правило вывода с действиями для обработки описаний будет таким:

$D \rightarrow$ ⟨ st_int.reset( ) ⟩ I ⟨ st_int.push (c_val) ⟩
{, I ⟨ st_int.push (c_val) ⟩ }:
[**int** ⟨ dec (LEX_INT) ⟩ | **bool** ⟨ dec (LEX_BOOL) ⟩]

Контроль контекстных условий в выражении

Типы операндов и обозначение операций мы будем хранить в стеке *Stack<type_of_lex, 100⟩ st_lex* .

Если в выражении встречается лексема целое_число или логические константы ***true*** или ***false***, то соответствующий тип сразу заносится в стек.

Если операнд — лексема-переменная, то необходимо проверить, описана ли она; если описана, то ее тип надо занести в стек. Эти действия можно выполнить с помощью функции *check_id*( ):

```
void parser::check_id()
{
    if ( TID[c_val].get_declare() )
        st_lex.push(TID[c_val].get_type());
    else
        throw "not declared";
}
```

Тогда для контроля контекстных условий каждой тройки — «операнд-операция-операнд» (для проверки соответствия типов операндов данной двуместной операции) будем использовать функцию *check_op*( ):

```
void Parser::check_op ()
{
    type_of_lex t1, t2, op, t = LEX_INT, r = LEX_BOOL;
    t2 = st_lex.pop();
    op = st_lex.pop();
    t1 = st_lex.pop();

    if ( op==LEX_PLUS || op==LEX_MINUS || op==LEX_TIMES || op==LEX_SLASH )
        r = LEX_INT;
    if ( op == LEX_OR || op == LEX_AND )
        t = LEX_BOOL;
    if ( t1 == t2 && t1 == t )
        st_lex.push(r);
    else
        throw "wrong types are in operation";
```

```
        prog.put_lex (Lex (op) );
    }
```

Для контроля за типом операнда одноместной операции ***not*** будем использовать функцию *check_not*( ):

```
    void Parser::check_not ()
    {
        if (st_lex.pop() != LEX_BOOL)
            throw "wrong type is in not";
        else
        {
            st_lex.push (LEX_BOOL);
            prog.put_lex (Lex (LEX_NOT));
        }
    }
```

Теперь главный вопрос: когда вызывать эти функции?

В грамматике модельного языка задано старшинство операций: наивысший приоритет имеет операция отрицания, затем в порядке убывания приоритета — группа операций умножения (∗, /, and), группа операций сложения (+, –, or), операции отношения.

$E \to E_1 \mid E_1\ [\ = \mid < \mid >\ ]\ E_1$

$E_1 \to T\ \{\ [+ \mid - \mid \textbf{\textit{or}}\ ]\ T\ \}$

$T \to F\ \{\ [\ * \mid / \mid \textbf{\textit{and}}\ ]\ F\}$

$F \to I \mid N \mid [\ \textbf{\textit{true}} \mid \textbf{\textit{false}}\ ] \mid \textbf{\textit{not}}\ F \mid (E)$

Именно это свойство грамматики позволит провести синтаксически-управляемый контроль контекстных условий.

Задача. Сравните грамматики, описывающие выражения, состоящие из символов +, ∗, (,), *i*:

G_1: $E \to E + E \mid E * E \mid (E) \mid I$

G_2: $E \to E + T \mid E * T \mid T$
$T \to i \mid (E)$

G_3: $E \to T + E \mid T * E \mid T$
$T \to i \mid (E)$

G_4: $E \to T \mid E + T$
$T \to F \mid T * F$
$F \to i \mid (E)$

G_5: $E \to T \mid T + E$
$T \to F \mid F * T$
$F \to i \mid (E)$

Оцените, насколько они удобны для трансляции выражений методом рекурсивного спуска (G_1 — неоднозначная, леворекурсивная; G_1, G_2, G_3 — не учитывается приоритет операций; G_4, G_5 — учитывается приоритет операций, G_2, G_4 — леворекурсивные, операции группируются слева направо, как принято в математике, G_3, G_5 — операции группируются справа налево).

Правила вывода выражений модельного языка с действиями для контроля контекстных условий:

$E \to E_1 \mid E_1\ [\ = \mid < \mid >\ ]\ \langle st\_char.push(TD[c\_val])\ \rangle\ E_1\ \langle check\_op(\ )\ \rangle$

$E_1 \to T\ \{\ [\ + \mid - \mid \textbf{\textit{or}}\ ]\ \langle st\_char.push\ (TD[c\_val]\ )\ \rangle\ T\ \langle check\_op(\ )\ \rangle\}$

$T \to F\ \{\ [\ * \mid / \mid \textbf{\textit{and}}\ ]\ \langle st\_char.push\ (TD[c\_val]\ )\ \rangle\ F\ \langle check\_op(\ )\ \rangle\}$

$F \rightarrow I \langle check\_id( ) \rangle \mid N \langle st\_char.push ("int") \rangle \mid$
$[true \mid false] \langle st\_char.push ("bool") \rangle \mid \textbf{not}\ F \langle check\_not( ) \rangle \mid (E)$

Контроль контекстных условий в операторах

$S \rightarrow I := E \mid \textbf{if}\ E\ \textbf{then}\ S\ \textbf{else}\ S \mid \textbf{while}\ E\ \textbf{do}\ S \mid B \mid read\ (I) \mid write\ (E)$

1. Оператор присваивания

$$I := E$$

Контекстное условие: в операторе присваивания типы переменной I и выражения E должны совпадать.

В результате контроля контекстных условий выражения E в стеке останется тип этого выражения (как тип результата последней операции); если при анализе идентификатора I проверить, описан ли он, и занести его тип в тот же стек (для этого можно использовать функцию *check_id*()), то достаточно будет в нужный момент считать из стека два элемента и сравнить их:

```
void Parser::eq_type ()
{
    if ( st_lex.pop() != st_lex.pop() )
        throw "wrong types are in :=";
}
```

Следовательно, правило для оператора присваивания:

$$I \langle \text{check\_id( )} \rangle := E \langle \text{eq\_type( )} \rangle$$

2. Условный оператор и оператор цикла

$$\textbf{if}\ E\ \textbf{then}\ S\ \textbf{else}\ S \mid \textbf{while}\ E\ \textbf{do}\ S$$

Контекстные условия: в условном операторе и в операторе цикла в качестве условия возможны только логические выражения.

В результате контроля контекстных условий выражения E в стеке останется тип этого выражения (как тип результата последней операции); следовательно, достаточно извлечь его из стека и проверить:

```
void Parser::eq_bool ()
{
    if ( st_lex.pop() != LEX_BOOL )
        throw "expression is not boolean";
}
```

Тогда правила для условного оператора и оператора цикла будут такими:

$$\textbf{if}\ E \langle \text{eq\_bool()} \rangle\ \textbf{then}\ S\ \textbf{else}\ S \mid \textbf{while}\ E \langle eq\_bool() \rangle\ \textbf{do}\ S$$

3. Для поверки операнда оператора ввода *read*(*I*) можно использовать следующую функцию:

```
void Parser::check_id_in_read ()
```

```
{
    if ( !TID [c_val].get_declare() )
        throw "not declared";
}
```

Правило для оператора ввода будет таким:

read (*I* ⟨ *check_id_in_read*( )⟩).

В итоге получаем процедуры для синтаксического анализа методом рекурсивного спуска с синтаксически-управляемым контролем контекстных условий, которые легко написать по правилам грамматики с действиями.

В качестве примера приведем функцию для нетерминала *D* (раздел описаний):

```
void Parser::D ()
{
    st_int.reset ();
    if ( c_type != LEX_ID )
        throw curr_lex;
    else
    {
        st_int.push ( c_val );
        gl();
        while (c_type == LEX_COMMA)
        {
            gl ();
            if (c_type != LEX_ID)
                throw curr_lex;
            else
            {
                st_int.push ( c_val );
                gl ();
            }
        }
        if ( c_type != LEX_COLON )
            throw curr_lex;
        else
        {
            gl ();
            if ( c_type == LEX_INT )
            {
                dec ( LEX_INT );
                gl();
            }
            else
            if ( c_type == LEX_BOOL )
            {
                dec ( LEX_BOOL );
                gl();
            }
            else
                throw curr_lex;
        }
    }
}
```

На этом мы заканчиваем рассмотрение фазы анализа интерпретатора модельного языка.

Генерация внутреннего представления программ

Результатом работы синтаксического анализатора должно быть некоторое внутреннее представление исходной цепочки лексем, которое отражает ее синтаксическую структуру. Программа в таком виде в дальнейшем может либо транслироваться в объектный код, либо интерпретироваться.

Язык внутреннего представления программы

Основные свойства языка внутреннего представления программ:

а) он позволяет фиксировать синтаксическую структуру исходной программы;

б) текст на нем можно автоматически генерировать во время синтаксического анализа;

в) его конструкции должны относительно просто транслироваться в объектный код либо достаточно эффективно интерпретироваться.

Некоторые общепринятые способы внутреннего представления программ:

а) постфиксная запись;

б) префиксная запись;

в) многоадресный код с явно именуемыми результатами;

г) многоадресный код с неявно именуемыми результатами;

д) связные списочные структуры, представляющие синтаксическое дерево.

В основе каждого из этих способов лежит некоторый метод представления синтаксического дерева.

Замечание: чаще всего синтаксическим деревом называют дерево вывода исходной цепочки, в котором удалены вершины, соответствующие цепным правилам вида $A \to B$, где $A, B \in N$.

Выберем в качестве языка для представления промежуточной программы ***постфиксную запись*** (ее часто называют ***ПОЛИЗ*** — польская инверсная запись).

ПОЛИЗ идеален для внутреннего представления интерпретируемых ЯП, которые, как правило, удобно переводятся в ПОЛИЗ и легко интерпретируются.

В ПОЛИЗе операнды выписаны слева направо в порядке их следования в исходном тексте. Знаки операций стоят таким образом, что знаку операции непосредственно предшествуют ее операнды.

Простым будем называть выражение, состоящее из одной константы или имени переменной. Такие выражения в ПОЛИЗе остаются без изменений. При переводе в ПОЛИЗ сложных выражений важно правильно определять границы подвыражений, являющихся левыми и правыми операндами бинарных операций. Проблем не возникает, если сложные подвыражения явно ограничены скобками. Например, в выражении $(a + b) * c$ левым операндом операции $*$ является подвыражение $a + b$, а правым — простое выражение c. Когда скобки явно не расставлены, как в случаях $a + b * c$ и $a - b + c$, важно учитывать *приоритет* операций, а также *ассоциативность* операций одинакового приоритета. Умножение имеет больший приоритет, чем сложение, поэтому в выражении $a + b * c$ операнд b относится к операции умножения и эквивалентное выражение со скобками будет таким: $a + (b * c)$. В выражении $a - b + c$ операнд b относится к

левой операции, т. е. к «минусу», а не к «плюсу» (в силу левой ассоциативности операций + и −, имеющих одинаковый приоритет), и это выражение эквивалентно выражению: $(a - b) + c$. Лево-ассоциативные операции группируются с помощью скобок слева направо: $a - b + c - d$ эквивалентно $((a - b) + c) - d$.

Теперь можем формально определить постфиксную запись выражений таким образом:

1) если E является простым выражением, то ПОЛИЗ выражения E — это само выражение E;
2) ПОЛИЗом выражения $E_1\ \theta\ E_2$, где θ — знак бинарной операции, E_1 и E_2 операнды для θ, является запись $E_1'\ E_2'\ \theta$, где E_1' и E_2' — ПОЛИЗ выражений E_1 и E_2 соответственно;
3) ПОЛИЗом выражения $\theta\ E$, где θ — знак унарной операции, а E — операнд θ, является запись $E'\ \theta$, где E' — ПОЛИЗ выражения E;
4) ПОЛИЗом выражения (E) является ПОЛИЗ выражения E.

Пример. ПОЛИЗом выражения $a + b + c$ является $a\ b + c +$, но не $a\ b\ c + +$. Последняя запись является ПОЛИЗом выражения $a + (b + c)$.

Алгоритм Дейкстры перевода в ПОЛИЗ выражений

Будем считать, что ПОЛИЗ выражения будет формироваться в массиве, содержащем лексемы — элементы ПОЛИЗа, и при переводе в ПОЛИЗ будет использоваться вспомогательный стек, также содержащий элементы ПОЛИЗа — операции, имена функций и круглые скобки.

1. Выражение просматривается один раз слева направо.
2. Пока есть непрочитанные лексемы входного выражения, выполняем действия:
   а) Читаем очередную лексему.
   б) Если лексема является числом или переменной, добавляем ее в ПОЛИЗ-массив.
   в) Если лексема является символом функции, помещаем ее в стек.
   г) Если лексема является разделителем аргументов функции (например, запятая), то до тех пор, пока верхним элементом стека не станет открывающаяся скобка, выталкиваем элементы из стека в ПОЛИЗ-массив. Если открывающаяся скобка не встретилась, это означает, что в выражении либо неверно поставлен разделитель, либо несогласованы скобки.
   д) Если лексема является операцией θ, тогда:
      - пока приоритет θ меньше либо равен приоритету операции, находящейся на вершине стека (для лево-ассоциативных операций), или приоритет θ строго меньше приоритета операции, находящейся на вершине стека (для право-ассоциативных операций) выталкиваем верхние элементы стека в ПОЛИЗ-массив;
      - помещаем операцию θ в стек.
   е) Если лексема является открывающей скобкой, помещаем ее в стек.
   ж) Если лексема является закрывающей скобкой, выталкиваем элементы из стека в ПОЛИЗ-массив до тех пор, пока на вершине стека не окажется открывающая скобка. При этом открывающая скобка удаляется

из стека, но в ПОЛИЗ-массив не добавляется. Если после этого шага на вершине стека оказывается символ функции, выталкиваем его в ПОЛИЗ-массив. Если в процессе выталкивания открывающей скобки не нашлось и стек пуст, это означает, что в выражении не согласованы скобки.

3. Когда просмотр входного выражения завершен, выталкиваем все оставшиеся в стеке символы в ПОЛИЗ-массив. (В стеке должны были оставаться только символы операций; если это не так, значит в выражении не согласованы скобки.)

Например, обычной (инфиксной) записи выражения

$$a * (b + c) - (d - e) / f$$

соответствует такая постфиксная запись:

$$a\ b\ c + *\ d\ e\ - f / -$$

Замечание: обратите внимание на то, что в ПОЛИЗе порядок операндов остался таким же, как и в инфиксной записи, учтено старшинство операций, а скобки исчезли.

Запись выражения в такой форме очень удобна для последующей интерпретации (т. е. вычисления значения этого выражения) с помощью стека.

Алгоритм вычисления выражений, записанных в ПОЛИЗе

1) Выражение просматривается один раз слева направо и для каждого элемента выполняются шаги (2) или (3);
2) Если очередной элемент ПОЛИЗа — операнд, то его значение заносится в стек;
3) Если очередной элемент ПОЛИЗа — операция, то на «верхушке» стека сейчас находятся ее операнды (это следует из определения ПОЛИЗа и предшествующих действий алгоритма); они извлекаются из стека, над ними выполняется операция, результат снова заносится в стек;
4) Когда выражение, записанное в ПОЛИЗе, прочитано, в стеке останется один элемент — это значение всего выражения, т. е. результат вычисления.

Замечание: для интерпретации, кроме ПОЛИЗа выражения, необходима дополнительная информация об операндах, хранящаяся в таблицах.

Замечание: может оказаться так, что знак бинарной операции по написанию совпадает со знаком унарной; например, знак "–" в большинстве языков программирования означает и бинарную операцию вычитания, и унарную операцию изменения знака. В этом случае во время интерпретации операции "–" возникнет неоднозначность: сколько операндов надо извлекать из стека и какую операцию выполнять. Устранить неоднозначность можно, по крайней мере, двумя способами:

- заменить унарную операцию бинарной, т. е. считать, что «–*a*» означает «0 − *a*»;
- либо ввести специальный знак для обозначения унарной операции; например, «–*a*» заменить на «&*a*». Важно отметить, что это изменение касается только внутреннего представления программы и не требует изменения входного языка.

Теперь необходимо разработать ПОЛИЗ для операторов входного языка. Каждый оператор языка программирования может быть представлен как *n*-местная операция с семантикой, соответствующей семантике этого оператора.

Оператор присваивания

$$I := E$$

в ПОЛИЗе будет записан как

$$\underline{I}\ E :=$$

где «:=» — это двухместная операция, а $\underline{I}$ и E — ее операнды; подчеркнутое $\underline{I}$ означает, что операндом операции «:=» является адрес переменной I, а не ее значение.

Оператор перехода в терминах ПОЛИЗа означает, что процесс интерпретации надо продолжить с того элемента ПОЛИЗа, который указан как операнд операции перехода.

Чтобы можно было ссылаться на элементы ПОЛИЗа, будем считать, что все они перенумерованы, начиная с 1 (допустим, занесены в последовательные элементы одномерного массива).

Пусть ПОЛИЗ оператора, помеченного меткой L, начинается с номера p, тогда оператор перехода *goto L* в ПОЛИЗе можно записать как

$$p\ !$$

где «!» — операция выбора элемента ПОЛИЗа, номер которого равен p.

Немного сложнее окажется запись в ПОЛИЗе **условных операторов и операторов цикла.**

Например, если рассматривать оператор *if B then* S_1 *else* S_2 как обычную трехместную операцию с операндами B, S_1, S_2, то ПОЛИЗ такого оператора должен выглядеть примерно так: $B'\ S_1'\ S_2'$ *if* , где B' — ПОЛИЗ условия, $S_1'\ S_2'$ — ПОЛИЗ операторов S_1, S_2, *if* — обозначение условной операции. Но тогда при интерпретации ПОЛИЗа обе ветви S_1, S_2 заранее вычисляются, независимо от условия B, что не соответствует семантике условного оператора. Для корректной реализации в ПОЛИЗе управляющих конструкций ***if***, ***while*** и т.п., их сначала заменяют эквивалентными фрагментами при помощи операторов перехода.

Введем вспомогательную операцию — условный переход «по лжи» с семантикой

$$\textbf{\textit{if}}\ (!B)\ \textbf{\textit{goto}}\ L$$

Это двухместная операция с операндами B и L. Обозначим ее $!F$, тогда в ПОЛИЗе она будет записана как

$$B'\ p\ !\mathrm{F}\ ,$$

где p — номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L, B' — ПОЛИЗ логического выражения B.

Семантика условного оператора

$$\textbf{\textit{if}}\ E\ \textbf{\textit{then}}\ S_1\ \textbf{\textit{else}}\ S_2$$

с использованием введенной операции может быть описана так:

$$\textbf{\textit{if}}\ (!\ E)\ \textbf{\textit{goto}}\ L_2;\ S_1;\ \textbf{\textit{goto}}\ L_3;\ L_2\colon S_2;\ L_3\colon \ldots$$

Тогда ПОЛИЗ условного оператора будет таким (порядок операндов — прежний!):

$$E'\ p_2\ !F\ S_1'\ p_3\ !\ S_2'\ \ldots\ ,$$

где p_i — номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L_i, $i = 2,3$, E' — ПОЛИЗ логического выражения E.

Семантика оператора цикла ***while*** E ***do*** S может быть описана так:

$$L_0\text{: } \textbf{\textit{if}}\ (!\ E)\ \textbf{\textit{goto}}\ L_1;\ S;\ \textbf{\textit{goto}}\ L_0;\ L_1\text{: } \ldots\ .$$

Тогда ПОЛИЗ оператора цикла ***while*** будет таким (порядок операндов — прежний!):

$$E'\ p_1\ !\mathrm{F}\ S'\ p_0\ !\ \ldots\ ,$$

где p_i — номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L_i, $i = 0,1$, E' — ПОЛИЗ логического выражения E.

Операторы ввода и вывода М-языка являются одноместными операциями.

Оператор ввода ***read*** (I) в ПОЛИЗе будет записан как $\underline{I}$ ***read*** ;

Оператор вывода ***write*** (E) в ПОЛИЗе будет записан как E' ***write*** , где E' — ПОЛИЗ выражения E.

Постфиксная польская запись операторов обладает всеми свойствами, характерными для постфиксной польской записи выражений, поэтому алгоритм интерпретации выражений пригоден для интерпретации всей программы, записанной на ПОЛИЗе (нужно только расширить набор операций; кроме того, выполнение некоторых из них не будет давать результата, записываемого в стек).

Постфиксная польская запись может использоваться не только для интерпретации промежуточной программы, но и для генерации по ней объектной программы. Для этого в алгоритме интерпретации вместо выполнения операции нужно генерировать соответствующие команды объектной программы.

Синтаксически управляемый перевод

На практике синтаксический, семантический анализ и генерация внутреннего представления программы часто осуществляются одновременно.

Существует несколько способов построения промежуточной программы. Один из них, называемый синтаксически управляемым переводом, особенно прост и эффективен.

В основе синтаксически управляемого перевода лежит уже известная нам грамматика с действиями (см. раздел о контроле контекстных условий). Теперь, параллельно с анализом исходной цепочки лексем, будем выполнять действия по генерации внутреннего представления программы. Для этого дополним грамматику вызовами соответствующих процедур генерации.

Содержательный пример — генерация внутреннего представления программы для М-языка — приведен ниже, а здесь в качестве иллюстрации рассмотрим более простой пример.

Пусть есть грамматика, описывающая простейшее арифметическое выражение. G_{expr} :

$$E \rightarrow T\ \{+T\ \}$$
$$T \rightarrow F\ \{*F\ \}$$
$$F \rightarrow a \mid b \mid (E)$$

Тогда грамматика с действиями по переводу этого выражения в ПОЛИЗ будет такой. G_{expr_polish}:

$$E \to T \{+T \langle cout << '+'; \rangle \}$$
$$T \to F \{*F \langle cout << '*'; \rangle \}$$
$$F \to a \langle cout << 'a'; \rangle \mid b \langle cout << 'b'; \rangle \mid (E)$$

В процессе анализа методом рекурсивного спуска входной цепочки $a + b * c$ по грамматике G_{expr_polish} в выходной поток будет выведена цепочка $a\ b\ c\ *\ +$, являющаяся польской записью исходной цепочки. Таким образом, данная грамматика с действиями каждой цепочке языка арифметических выражений ставит в соответствие подходящую цепочку языка польских записей арифметических выражений. Иными словами, такая грамматика задает *перевод* из одного формального языка в другой.

Определение: Пусть Σ и Δ — алфавиты. *Формальный перевод* T — это подмножество множества всевозможных пар цепочек в алфавитах Σ и Δ : $T \subseteq (\Sigma^* \times \Delta^*)$.

Назовем *входным* языком перевода T язык $L_{вх} = \{\alpha \mid \exists\ \beta : (\alpha, \beta) \in T \}$.

Назовем *целевым* (или *выходным*) языком перевода T язык $L_{ц} = \{\beta \mid \exists\ \alpha : (\alpha, \beta) \in T \}$.

Перевод T *неоднозначен*, если для некоторых $\alpha \in \Sigma^*$, $\beta, \gamma \in \Delta^*$, $\beta \neq \gamma$ $(\alpha, \beta) \in T$ и $(\alpha, \gamma) \in T$.

Рассмотренная выше грамматика G_{expr_polish} задает однозначный перевод: каждому выражению ставится в соответствие единственная польская запись. Неоднозначные переводы могут быть интересны при изучении моделей естественных языков; для трансляции языков программирования используются однозначные переводы.

Заметим, что для двух заданных языков L_1 и L_2 существует бесконечно много формальных переводов[23]. Чтобы задать перевод из L_1 в L_2, важно точно указать *закон соответствия* между цепочками L_1 и L_2.

Пример. Пусть $L_1 = \{ 0^n 1^m \mid n \geq 0, m > 0 \}$ — входной язык, $L_2 = \{a^m b^n \mid n \geq 0, m > 0\}$ — выходной язык и перевод T определяется так: для любых $n \geq 0$, $m > 0$ цепочке $0^n 1^m \in L_1$ соответствует цепочка $a^m b^n \in L_2$. Можно записать T с помощью теоретико-множественной формулы: $T = \{ (0^n 1^m, a^m b^n) \mid n \geq 0, m > 0 \}$, $L_{вх} = L_1$, $L_{ц} = L_2$.

Задача

Требуется реализовать перевод T грамматикой с действиями.

Решение

Язык L_1 можно описать грамматикой:

$$S \to 0S \mid 1A$$
$$A \to 1A \mid \varepsilon$$

Вставим действия по переводу цепочек вида $0^n 1^m$ в соответствующие цепочки вида $a^m b^n$:

$$S \to 0S \langle cout << 'b'; \rangle \mid 1 \langle cout << 'a'; \rangle A$$
$$A \to 1 \langle cout << 'a'; \rangle A \mid \varepsilon$$

[23] При условии, что хотя бы один из языков L_1 , L_2 бесконечен.

Теперь при анализе методом рекурсивного спуска цепочек языка L_1 с помощью действий будут порождаться соответствующие цепочки языка L_2.

Генератор внутреннего представления программы на М-языке

Каждый элемент в ПОЛИЗе — это лексема, т. е. пара вида ⟨ тип_лексемы, значение_лексемы ⟩. При генерации ПОЛИЗа будем использовать дополнительные типы лексем:

- *POLIZ_GO* – «!»;
- *POLIZ_FGO* – «!F»;
- *POLIZ_LABEL* – для ссылок на номера элементов ПОЛИЗа;
- *POLIZ_ADDRESS* – для обозначения операндов-адресов (например, в ПОЛИЗе оператора присваивания).

Будем считать, что генерируемая программа размещается в объекте *Poliz prog* (1000); класса *Poliz*:

```
class Poliz
{
    lex *p;
    int size;
    int free;
public:
    Poliz (int max_size )
    {
        p=new Lex [size = max_size];
        free = 0;
    };
    ~Poliz() { delete []p; };
    void put_lex(Lex l) { p[free]=l; free++; };
    void put_lex(Lex l, int place) { p[place]=l; };
    void blank() { ++free; };
    int get_free() { return free; };
    lex& operator[](int index)
    {
        if (index > size)
            throw "POLIZ:out of array";
        else
            if(index > free)
                throw "POLIZ:indefinite element of array";
            else
                return p[index];
    };
    void print()
    {
        for (int i=0; i < free; i++)
            cout << p[i];
    };

};
```

Генерация внутреннего представления программы будет проходить во время синтаксического анализа параллельно с контролем контекстных условий, поэтому для генерации можно использовать информацию, «собранную» синтаксическим и семанти-

ческим анализаторами; например, при генерации ПОЛИЗа выражений можно воспользоваться содержимым стека, с которым работает семантический анализатор.

Кроме того, можно дополнить функции семантического анализа действиями по генерации (вспомните функции *check_op*() и *check_not*()).

Тогда грамматика, содержащая действия по контролю контекстных условий и переводу выражений модельного языка в ПОЛИЗ, будет такой:

$E \rightarrow E_1 \mid E_1 [= \mid < \mid >] \langle st\_lex.push (TD [c\_val]) \rangle E_1 \langle check\_op ( ) \rangle$

$E_1 \rightarrow T \{ [+ \mid - \mid \textbf{or}] \langle st\_lex.push (TD [c\_val]) \rangle T \langle check\_op ( ) \rangle \}$

$T \rightarrow F \{ [* \mid / \mid \textbf{and}] \langle st\_lex.push (TD [c\_val]) \rangle F \langle check\_op ( ) \rangle \}$

$F \rightarrow I \langle check\_id ( ); prog.put\_lex (curr\_lex); \rangle \mid$

$N \langle st\_lex.push (LEX\_INT); prog.put\_lex (curr\_lex); \rangle \mid$

$[\textbf{true} \mid \textbf{false}] \langle st\_lex.push (LEX\_BOOL); prog.put\_lex (curr\_lex); \rangle \mid$

$\textbf{not}\ F \langle check\_not( ); \rangle \mid (E)$

Действия, которыми нужно дополнить правило вывода оператора присваивания, также достаточно очевидны:

$S \rightarrow I \langle check\_id ( ); prog.put\_lex (Lex (POLIZ\_ADDRESS, c\_val)); \rangle :=$

$E \langle eqtype ( ); prog.put\_lex (Lex (LEX\_ASSIGN)); \rangle$

При генерации ПОЛИЗа выражений и оператора присваивания элементы объекта ***prog*** заполнялись последовательно. Семантика условного оператора ***if*** E ***then*** S_1 ***else*** S_2 такова, что значения операндов для операций безусловного перехода и перехода «по лжи» в момент генерации операций еще неизвестны:

$$\textbf{if}\ (!E)\ \textbf{goto}\ l_2;\ S_1;\ \textbf{goto}\ l_3;\ l_2:\ S_2;\ l_3 :\ldots$$

Поэтому придется запоминать номера элементов объекта ***prog***, соответствующих этим операндам, а затем, когда станут известны их значения, заполнять пропущенное.

Тогда грамматика, содержащая действия по контролю контекстных условий и переводу условного оператора модельного языка в ПОЛИЗ, будет такой:

$S \rightarrow \textbf{if}\ E \langle \text{eqbool( )}; pl_2 = prog.get\_free( ); prog.blank( );$

$prog.put\_lex (Lex (POLIZ\_FGO)); \rangle$

$\textbf{then}\ S_1 \langle pl_3 = prog.get\_free( ); prog.blank( );$

$prog.put\_lex (Lex (POLIZ\_GO));$

$prog.put\_lex (Lex (POLIZ\_LABEL, prog.get\_free( )), pl_2); \rangle$

$\textbf{else}\ S_2 \langle prog.put\_lex (Lex (POLIZ\_LABEL, prog.get\_free( )), pl_3); \rangle$

Семантика оператора цикла **while** E **do** S описывается так:

L_0: ***if*** $(!E)$ ***goto*** l_1; S; ***goto*** l_0; l_1: ... ,

а грамматика с действиями по контролю контекстных условий и переводу оператора цикла в ПОЛИЗ будет такой:

$S \rightarrow \textbf{while} \langle pl_0 = prog.get\_free ( ); \rangle E \langle eqbool ( );$

$pl_1 = prog.get\_free ( ); prog.blank ( );$

$prog.put\_lex (Lex (POLIZ\_FGO)); \rangle$

$\textbf{do}\ S \langle prog.put\_lex (Lex (POLIZ\_LABEL, pl_0);$

$prog.put\_lex (Lex (POLIZ\_GO));$

$prog.put\_lex (Lex (POLIZ\_LABEL, prog.get\_free( )), pl_1); \rangle$

Замечание: переменные pl_i ($i = 0, 1, 2, 3$) должны быть локализованы в процедуре *S*, иначе возникнет ошибка при обработке вложенных условных операторов.

Грамматика с действиями по контролю контекстных условий и переводу в ПОЛИЗ операторов ввода и вывода будет такой:

$S \rightarrow read$ (I ⟨ *check_id_in_read*( );
prog.put_lex (*Lex* (*POLIZ_ADDRESS*, *c_val*)); ⟩)
⟨ *prog.put_lex* (*Lex* (*LEX_READ*)); ⟩
$S \rightarrow write$ (E) ⟨ *prog.put_lex* (*Lex* (*LEX_WRITE*)); ⟩

Интерпретатор ПОЛИЗа для модельного языка

Польская инверсная запись была выбрана нами в качестве языка внутреннего представления программы, в частности, потому, что записанная таким образом программа может быть легко проинтерпретирована.

Идея алгоритма очень проста: просматриваем ПОЛИЗ слева направо; если встречаем операнд, то записываем его в стек; если встретили знак операции, то извлекаем из стека нужное количество операндов и выполняем операцию, результат (если он есть) заносим в стек и т. д.

Итак, программа на ПОЛИЗе хранится в виде последовательности лексем в объекте *prog* класса *Poliz*. Лексемы могут быть следующие: лексемы-константы (числа, *true*, *false*), лексемы-метки ПОЛИЗа, лексемы-операции (включая введенные в ПОЛИЗе) и лексемы-переменные (их значения или адреса — номера строк в таблице *TID*).

В программе-интерпретаторе будем использовать некоторые переменные и функции, введенные нами ранее.

```
class Executer
{
    Lex pc_el;
public:
    void execute ( Poliz& prog );
};

void Executer::execute ( Poliz& prog )
{
    Stack < int, 100 > args;
    int i, j, index = 0, size = prog.get_free();

    while ( index < size )
    {
        pc_el = prog [ index ];
        switch ( pc_el.get_type () )
        {
            case LEX_TRUE:
            case LEX_FALSE:
            case LEX_NUM:
            case POLIZ_ADDRESS:
            case POLIZ_LABEL:
                args.push ( pc_el.get_value () );
                break;
            case LEX_ID:
                i = pc_el.get_value ();
                if ( TID[i].get_assign () )
                {
                    args.push ( TID[i].get_value () );
```

```
                    break;
                }
                else
                    throw "POLIZ: indefinite identifier";
            case LEX_NOT:
                args.push( !args.pop() );
                break;
            case LEX_OR:
                i = args.pop();
                args.push ( args.pop() || i );
                break;
            case LEX_AND:
                i = args.pop();
                args.push ( args.pop() && i );
                break;
            case POLIZ_GO:
                index = args.pop() - 1;
                break;
            case POLIZ_FGO:
                i = args.pop();
                if ( !args.pop() )
                    index = i-1;
                break;
            case LEX_WRITE:
                cout << args.pop () << endl;
                break;
            case LEX_READ:
                {
                    int k;
                    i = args.pop ();
                    if ( TID[i].get_type () == LEX_INT )
                    {
                        cout << "Input int value for";
                        cout << TID[i].get_name () << endl;
                        cin >> k;
                    }
                    else
                    {
                        char j[20];
                    rep:
                        cout << "Input boolean value;
                        cout << (true or false) for";
                        cout << TID[i].get_name() << endl;
                        cin >> j;
                        if ( !strcmp(j, "true") )
                              k = 1;
                        else if ( !strcmp(j, "false") )
                              k = 0;
                        else
                        {
                              cout << "Error in input:true/false";
                              cout << endl;
                              goto rep;
                        }
                }
                TID[i].put_value (k);
                TID[i].put_assign ();
                break;}
            case LEX_PLUS:
                args.push ( args.pop() + args.pop() );
```

```
                    break;
                case LEX_TIMES:
                    args.push ( args.pop() * args.pop() );
                    break;
                case LEX_MINUS:
                    i = args.pop();
                    args.push ( args.pop() - i );
                     break;
                case LEX_SLASH:
                    i = args.pop();
                    if ( !i )
                    {
                         args.push (args.pop() / i);
                         break;
                    }
                    else throw "POLIZ:divide by zero";
                case LEX_EQ:
                    args.push ( args.pop() == args.pop() );
                    break;
                case LEX_LSS:
                    i = args.pop();
                    args.push ( args.pop() < i);
                    break;
                case LEX_GTR:
                    i = args.pop();
                    args.push ( args.pop() > i );
                    break;
                case LEX_LEQ:
                    i = args.pop();
                    args.push ( args.pop() <= i );
                    break;
                case LEX_GEQ:
                    i = args.pop();
                    args.push ( args.pop() >= i );
                    break;
                case LEX_NEQ:
                    i = args.pop();
                    args.push ( args.pop() != i );
                    break;
                case LEX_ASSIGN:
                    i = args.pop();
                    j = args.pop();
                    TID[j].put_value(i);
                    TID[j].put_assign();
                    break;
                default:
                    throw "POLIZ: unexpected elem";
            }                           // end of switch
            ++index;
        };                              //end of while
        cout << "Finish of executing!!!" << endl;
    }

    class Interpretator
    {
        Parser pars;
        Executer E;
    public:
        Interpretator ( char* program ): pars (program) {};
        void interpretation ();
```

```
};

void Interpretator::interpretation ()
{
    pars.analyze ();
    E.execute ( pars.prog );
}

int main ()
{
    try
    {
        Interpretator I ( "program.txt" );
        I.interpretation ();
        return 0;
    }
    catch ( char c )
    {
        cout << "unexpected symbol " << c << endl;
        return 1;
    }
    catch ( Lex l )
    {
        cout << "unexpected lexeme";
        cout << l;
        return 1;
    }
    catch ( const char *source )
    {
        cout << source << endl;
        return 1;
    }
}
```

Литература

[1]. Д. Грис. Конструирование компиляторов для цифровых вычислительных машин. — М.: Мир, 1975.

[2]. Ф. Льюис, Д. Розенкранц, Р. Стирнз. Теоретические основы проектирования компиляторов. — М.: Мир, 1979.

[3]. А. Ахо, Дж. Ульман. Теория синтаксического анализа, перевода и компиляции. — Т. 1,2. — М.: Мир, 1979.

[4]. Ф. Вайнгартен. Трансляция языков программирования. — М.: Мир, 1977.

[5]. И. Л. Братчиков. Синтаксис языков программирования. — М.: Наука, 1975.

[6]. С. Гинзбург. Математическая теория контекстно-свободных языков. — М.: Мир, 1970.

[7]. Дж. Фостер. Автоматический синтаксический анализ. — М.: Мир, 1975.

[8]. В. Н. Лебедев. Введение в системы программирования. — М.: Статистика, 1975.

[9]. Б. Ф. Мельников. Подклассы класса контекстно-свободных языков. — М.: МГУ, 1995.

[10]. В. Н. Пильщиков, В. Г. Абрамов, А. А. Вылиток, И. В. Горячая. Машины Тьюринга и алгоритмы Маркова. Решение задач. — М.:МГУ, 2006.

[11]. А. Ахо., Р. Сети, Дж. Ульман. Компиляторы: принципы, технологии, инструменты. — М.: «Вильямс», 2001.

[12]. А. Ахо, М. Лам, Р.Сети, Дж. Ульман. Компиляторы: принципы, технологии и инструментарий. — М.: «Вильямс», 2008.

Содержание