ЭФФЕКТИВНЫЙ АЛГОРИТМ АНАЛИЗА КОНТЕКСТНО-СВОБОДНЫХ ЯЗЫКОВ [1])

Джей Эрли

1. Введение

Контекстно-свободные грамматики (грамматики Бэкуса) широко используются для описания синтаксиса языков программирования и естественных языков. В силу этого алгоритмы анализа контекстно-свободных грамматик играют большую роль при разработке компиляторов и интерпретаторов для языков программирования, а также программ для «чтения» («понимания») и перевода естественных языков.

Было предложено много алгоритмов анализа. Некоторые являются общими, т. е. они пригодны для любых контекстно-свободных грамматик, тогда как другие оперируют лишь на подклассах класса контекстно-свободных грамматик. Последние, ограниченные алгоритмы оказываются обычно намного более эффективными. Алгоритм, описываемый в этой статье, по-видимому, наиболее эффективен среди всех общих алгоритмов. Кроме того, он работает в линейное время на более широком классе грамматик, чем большинство известных ограниченных алгоритмов. Эффективность алгоритмов мы оцениваем как с помощью формального исследования, так и с помощью их эмпирического сравнения.

Эта статья основана на докладе автора 1968 г. [1], где многие представленные здесь вопросы рассматриваются значительно подробнее. В разд. 2 определяются термины, используемые в статье. В разд. 3 алгоритм описывается неформально, а в разд. 4 дается точное описание. Раздел 5 посвящен изучению формальных свойств эффективности, и читатель, не интересующийся этим аспектом, может его пропустить. В разд. 6 проводится эмпирическое сравнение алгоритмов, а в разд. 7 обсуждается практическое применение предложенного алгоритма.

2. Терминология

Языком называется множество цепочек над алфавитом, состоящим из конечного числа символов. Эти символы мы будем

[1]) Earley J., An efficient context-free parsing algorithm, *Commun. ACM*, 13, 2 (1970), 94—102.

называть *терминальными* и обозначать строчными латинскими буквами a, b, c, ...

Контекстно-свободную грамматику мы будем использовать как средство для определения того, какие цепочки принадлежат этому множеству. В ней применяется еще и другое множество символов, называемых *нетерминалами*, которые можно рассматривать как синтаксические классы. Их мы будем обозначать прописными латинскими буквами A, B, C, ... Цепочки из терминалов и нетерминалов обозначаются греческими буквами α, β, γ, ... Пустая цепочка обозначается через λ, α^k — цепочка $\underbrace{\alpha \ldots \alpha}_{k \text{ раз}}$, $|\alpha|$ — число символов в цепочке α, или ее длина. Имеется конечное число *продукций*, или *правил подстановки*, $A \to \alpha$. Нетерминал, соответствующий классу «предложение», называется *корнем* (*аксиомой*, *начальным символом*) R грамматики. Продукции с фиксированным нетерминалом D слева называются *альтернативами* символа D. Везде в этой статье грамматика будет контекстно-свободной.

Рассмотрим в качестве примера грамматику AE простых арифметических выражений:

$$E \to I$$
$$E \to E + T$$
$$T \to P$$
$$T \to T * P$$
$$P \to a$$

Терминальными символами ее являются $\{a, +, *\}$, нетерминальными — $\{E, T, P\}$, а E — корень.

Бо́льшая часть оставшихся определений относится к фиксированной грамматике G. Мы будем писать $\alpha \Rightarrow \beta$, если найдутся такие γ, δ, η, A, что $\alpha = \gamma A \delta$, $\beta = \gamma\eta\delta$ и $A \to \eta$ есть продукция; мы будем писать $\alpha \Rightarrow^* \beta$ и говорить, что цепочка β *выводится* из α, если найдутся такие цепочки $\alpha_0, \alpha_1, \ldots, \alpha_m$ $(m \geqslant 0)$, что

$$\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_m = \beta.$$

Последовательность $\alpha_0, \ldots, \alpha_m$ называется *выводом* (цепочки β из α).

Выводимой цепочкой называется такая цепочка α, что $R \Rightarrow^* \alpha$. *Предложение* — это выводимая цепочка, целиком состоящая из терминальных символов. *Языком* $L(G)$, *определенным грамматикой* G, называется множество всех ее предложений. Каждую выводимую цепочку можно по крайней мере одним способом представить в виде *дерева вывода* (или *дерева анализа*), отражающего шаги, проделанные при его выводе (но

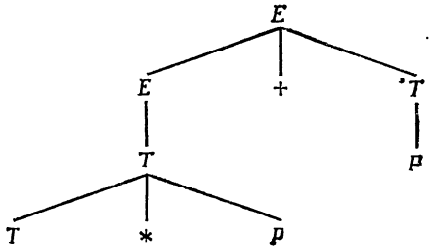
не отражающего порядок этих шагов). Например, для грамматики AE оба вывода

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow T + P \Rightarrow T * P + P$$

и

$$E \Rightarrow E + T \Rightarrow E + P \Rightarrow T + P \Rightarrow T * P + P$$

представляются одним деревом



Степенью неоднозначности предложения называется число его различных деревьев вывода. Предложение называется *однозначным*, если степень его неоднозначности равна 1. Грамматика называется *однозначной*, если все ее предложения однозначны. Грамматика называется *грамматикой ограниченной неоднозначности*, если существует общая граница b степеней неоднозначности всех предложений грамматики. Грамматика называется *приведенной*, если каждый нетерминал появляется в некотором выводе некоторого предложения.

Распознавателем называется алгоритм, работающий над цепочкой, который *допускает* или *отвергает* ее в зависимости от того, является или нет эта цепочка предложением грамматики. *Анализатором* называется распознаватель, который также выдает все возможные деревья выводов для цепочки.

3. Неформальное изложение

Опишем неформально алгоритм распознавания: входная цепочка $X_1 \ldots X_n$ читается слева направо, при этом происходит считывание вперед на фиксированное число k символов. При считывании каждого символа X_i строится множество положений S_i, дающее условие для распознавания в этот момент. Каждое положение из этого множества представляет: (1) правило, согласно которому в данный момент считывается сегмент входной цепочки, выводимый из правой части правила; (2) место в этом правиле, показывающее, какая доля правой части правила уже распознана; (3) указатель возврата к позиции на входной цепочке, с которой мы начали искать возможность применения этого правила; (4) k-символьную цепочку, являющуюся

синтаксически допустимым преемником применяемого правила. Эта четверка представлена здесь как правило с точкой в нем, за которой следует натуральное число и некоторая цепочка.

Например, если бы при распознавании порождаемости цепочки $a * a$ в грамматике АЕ мы прочитали первый символ a, то пришли бы к следующему множеству положений S_1 (исключая k-символьные цепочки):

$$\begin{array}{ll} P \to a. & 0 \\ T \to P. & 0 \\ T \to T. * P & 0 \\ E \to T. & 0 \\ E \to E. + T & 0 \end{array}$$

Каждое положение представляет возможный анализ для начала цепочки при условии, что прочитан только символ a. Все положения имеют в качестве указателя 0, поскольку все представленные правила должны начинаться с начала цепочки.

В цепочке для каждой позиции будет одно такое множество положений. Для облегчения распознавания поместим $k+1$ правых граничных маркеров $\dashv$ (символ, который не встречается ни в каком другом месте грамматики) у правого конца входной цепочки.

В начале работы алгоритма полагаем, что множество положений S_0 содержит единственное положение

$$\varnothing \to .\, R \dashv \quad \dashv^k \quad 0,$$

где R — корень грамматики, а $\varnothing$ — новый нетерминал.

Вообще мы действуем на множестве положений S_i следующим образом: по порядку к каждому положению в зависимости от его вида применяется одна из трех операций. Эти операции могут добавлять новые положения к S_i и могут вводить положения в новое множество положений S_{i+1}. Опишем эти три операции на примере.

В грамматике АЕ с $k = 1$ множество S_0 содержит единственное положение

$$\varnothing \to .\, E \dashv \quad \dashv \quad 0. \tag{1}$$

Операцию, применимую к положению, в котором справа от точки стоит нетерминал, называют *предсказателем*. Для каждой альтернативы этого нетерминала предсказатель заставляет добавить одно новое положение к S_i. В каждом новом положении поставим точку в начале правила, поскольку в новом положении мы не прочитали еще ни одного символа. Указатель направим к i, поскольку это новое положение образовано в S_i.

Таким образом, предсказатель добавляет к S_i все правила, способные порождать подцепочки, начинающиеся с X_{i+1}.

В нашем примере к S_0 добавляются положения

$$E \rightarrow .\, E + T \dashv 0 \qquad (2)$$

$$E \rightarrow .\, T \dashv 0 \qquad (3)$$

Цепочка из k символов, считываемая вперед, есть $\dashv$, так как в начальном положении она стоит после E. Нам надо обработать эти два положения. К ним можно применить предсказатель. Действуя на (2), он дает

$$E \rightarrow .\, E + T + 0 \qquad (4)$$

$$E \rightarrow .\, T + 0 \qquad (5)$$

со считываемым вперед символом $+$, поскольку в (2) этот символ стоит после E. Действуя на (3), предсказатель дает

$$T \rightarrow .\, T * P \dashv 0$$

$$T \rightarrow .\, P \dashv 0$$

Здесь считываемый вперед символ есть $\dashv$, поскольку T — последний символ в правиле, а $\dashv$ — считываемый вперед символ для (3). Далее, действуя на (4), предсказатель дает снова (4) и (5), но они уже содержатся в S_0, так что ничего не изменяется. Из (5) получается

$$T \rightarrow .\, T * P + 0$$

$$T \rightarrow .\, P + 0$$

Оставшаяся часть множества S_0 такова:

$$T \rightarrow .\, T * P * 0$$

$$T \rightarrow .\, P * 0$$

$$P \rightarrow .\, a \dashv 0$$

$$P \rightarrow .\, a + 0$$

$$P \rightarrow .\, a * 0$$

Ни к одному из последних трех положений предсказатель применить нельзя. Вместо него действует *считыватель*, применимый как раз тогда, когда справа от точки стоит терминал. Считыватель сравнивает этот терминал с X_{i+1}, и, если они совпадают, это положение добавляется к S_{i+1}, причем точка в этом положении сдвигается на 1, указывая, что прочитан очередной терминальный символ.

Если $X_1 = a$, то S_1 есть

$$\begin{array}{l} P \to a. \dashv 0 \\ P \to a. + 0 \\ P \to a. * 0 \end{array} \qquad (6)$$

причем эти положения добавлены считывателем.

Допустим, что мы закончили[1]) обработку множества S_i, а множество S_{i+1} осталось пустым. Это означает, что во входной цепочке встретилась ошибка. Если же множество S_{i+1} не пусто, мы начинаем его обработку.

Третья операция, *восполнитель*, применима к положению, когда точка находится в конце правила. Таким образом, восполнитель применим к каждому из трех положений в S_1. Он сравнивает считываемую вперед цепочку с $X_{i+1} \ldots X_{i+k}$. Если они совпадают, он возвращается назад к множеству положений, определяемому указателем, в данном случае к S_0, выбирает все положения из S_0, у которых справа от точки стоит P, передвигает в этих положениях точку через P и все полученные таким образом положения добавляет к S_1. Интуитивно: S_0 — это множество положений, в котором мы были, когда искали нетерминал P. Когда мы его находим, мы возвращаемся ко всем тем положениям в S_0, которые заставили нас искать P, и сдвигаем точку через P в этих положениях, чтобы показать, что порождение нетерминала P уже успешно прочитано.

Если $X_2 = +$, то восполнитель, примененный к (6), заставляет нас добавить к S_1

$$\begin{array}{l} T \to P. \dashv 0 \\ T \to P. + 0 \\ T \to P. * 0 \end{array}$$

Применение восполнителя ко второму из этих положений дает

$$\begin{array}{ll} E \to T. & \dashv 0 \\ E \to T. & + 0 \\ T \to T. * P & \dashv 0 \\ T \to T. * P & + 0 \\ T \to T. * P & * 0 \end{array}$$

и, наконец, из второго положения последней группы получаем

$$\begin{array}{ll} \varnothing \to E. \dashv & \dashv 0 \\ E \to E. + T & \dashv 0 \\ E \to E. + T & + 0 \end{array}$$

[1]) То есть применение трех указанных правил больше не приводит к увеличению числа положений в S_i. — *Прим. ред.*

Считыватель теперь добавляет к S_2

$$E \to E + .T \dashv 0$$
$$E \to E + .T + 0$$

Если алгоритм когда-нибудь построит множество S_{i+1}, состоящее из единственного положения

$$\varnothing \to E \dashv . \quad \dashv 0$$

это будет означать, что мы прочитали правильное выражение E, завершаемое символом $\dashv$, так что работа над цепочкой закончена и эта цепочка является предложением нашей грамматики.

Полный список действий алгоритма на грамматике АЕ приведен в табл. 1. В этом примере мы записали все положения из одного множества положений, отличающиеся только своей считываемой вперед цепочкой, в виде одного положения. (Таким

Таблица 1

ГРАММАТИКА АЕ

корень: $E \to T \mid E + T$ входная цепочка $= a + a*a$
$T \to P \mid T*P$
$P \to a$

			$k=1$				
S_0	$\varnothing \to .E\dashv$	$\dashv$	0	S_3	$P \to a.$	$\dashv + *$	2
$(X_1 = a)$	$E \to .E + T$	$\dashv +$	0	$(X_4 = *)$	$T \to P.$	$\dashv + *$	2
	$E \to .T$	$\dashv +$	0		$E \to E + T.$	$\dashv +$	0
	$T \to .T*P$	$\dashv + *$	0		$T \to T.*P$	$\dashv + *$	2
	$T \to .P$	$\dashv + *$	0				
	$P \to .a$	$\dashv + *$	0	S_4	$T \to T.*P$	$\dashv + *$	2
				$(X_5 = a)$	$P \to .a$	$\dashv + *$	4
S_1	$P \to a.$	$\dashv + *$	0				
$(X_2 = +)$	$T \to P.$	$\dashv + *$	0	S_5	$P \to a.$	$\dashv + *$	4
	$E \to T.$	$\dashv +$	0	$(X_6 = \dashv)$	$T \to T*P.$	$\dashv + *$	2
	$T \to T.*P$	$\dashv + *$	0		$E \to E + T.$	$\dashv +$	0
	$\varnothing \to E.\dashv$	$\dashv$	0		$T \to T.*P$	$\dashv + *$	2
	$E \to E.+T$	$\dashv +$	0		$\varnothing \to E.\dashv$	$\dashv$	0
					$E \to E.+T$	$\dashv +$	0
S_2	$E \to E + .T$	$\dashv +$	0				
$(X_3 = a)$	$T \to .T*P$	$\dashv + *$	2	S_6	$\varnothing \to E\dashv .$	$\dashv$	0
	$T \to .P$	$\dashv + *$	2				
	$P \to .a$	$\dashv + *$	2				

образом, $\dashv + *$ как считываемая вперед цепочка обозначает три положения с $\dashv$, $+$ и $*$ в качестве соответствующих считываемых вперед цепочек.)

Техника использования множеств положений и считывания вперед взята из работы Кнута об LR(k)-грамматиках [2]. Действительно, наш алгоритм тесно связан с алгоритмом Кнута для LR(k)-грамматик; единственное отличие его от алгоритма Кнута состоит в том, что для запоминания пути, который будет выбран после того, как будет произведено некоторое распознавание, он использует стек, а не указатели.

Заметим также, что наш алгоритм, хотя он и не осуществляется сверху вниз, является по существу нисходящим анализатором [3], в котором все возможные анализы проводятся совместно и таким образом, что похожие поданализы можно часто комбинировать. Это снижает дублирование усилий и позволяет избежать проблемы левой рекурсии. («Прямолинейный» нисходящий анализатор на грамматике, содержащей левую рекурсию, т. е. $A \to A\beta$, может войти в бесконечный цикл.)

4. Распознаватель

Приведем теперь точное описание работы алгоритма распознавания для входной цепочки $X_1 \ldots X_n$ и грамматики G.

Обозначение. Занумеруем правила грамматики G произвольно числами 1, ..., $d-1$; каждое правило имеет вид

$$D_p \to C_{p1} \ldots C_{p\bar{p}} \qquad (1 \leqslant p \leqslant d-1),$$

где $\bar{p}$ — число символов в правой части p-го правила. Добавим 0-е правило $D_0 \to R \dashv$, где R — аксиома грамматики G, а $\dashv$ — новый терминальный символ.

Определение 4.1. *Положением* называется четверка $\langle p, j, f, \alpha\rangle$, где p, j и f — целые числа ($0 \leqslant p \leqslant d-1$, $0 \leqslant j \leqslant \bar{p}$, $0 \leqslant f \leqslant n+1$), а α — цепочка, состоящая из k терминалов. *Множеством положений* будем называть упорядоченное множество, элементами которого являются положения. *Заключительное положение* — это положение, для которого $j = \bar{p}$. Мы *добавляем* положение к множеству положений, помещая его последним в упорядоченном множестве, *если только оно еще не входит в это множество.*

Определение 4.2. $H_k(\gamma) = \{\alpha \mid \alpha$ — терминальная цепочка, $|\alpha| = k$ и $\exists\beta(\gamma \Rightarrow {}^*\alpha\beta)\}$.

$H_k(\gamma)$ — это множество всех k-символьных терминальных цепочек, с которых начинаются цепочки, выводимые из γ. Оно

используется при построении считываемой вперед цепочки для положений.

Распознавателем называется функция REC(G, $X_1 \ldots X_n$, k) трех аргументов, вычисляемая следующим образом:

Пусть $X_{n+i} = \dashv$ $(1 \leqslant i \leqslant k+1)$.
Множество S_i $(0 \leqslant i \leqslant n+1)$ пусто.
Добавим $\langle 0, 0, 0, \dashv^k \rangle$ к S_0.

Для i от 0, прибавляя по 1 вплоть до n, выполнить
Начало

Положения из множества S_i обрабатываются по порядку, при этом на каждом положении $s = \langle p, j, f, \alpha \rangle$ совершается одна из трех следующих операций.

(1) Предсказатель: если положение s не является заключительным, а $C_{p(j+1)}$ — нетерминал, то для каждого q, для которого $C_{p(j+1)} = D_q$, и для каждого $\beta \in H_k(C_{p(j+2)} \ldots C_{p\bar{p}}\alpha)$ добавляем к S_i положение $\langle q, 0, i, \beta \rangle$.

(2) Восполнитель: если s — заключительное положение и $\alpha = X_{i+1} \ldots X_{i+k}$, то для каждого положения $\langle q, l, g, \beta \rangle \in S_f$ (после того, как все положения были добавлены к S_f), для которого $C_{q(l+1)} = D_p$, добавляем к S_i положение $\langle q, l+1, g, \beta \rangle$.

(3) Считыватель: если положение s не является заключительным и $C_{p(j+1)}$ — терминал, то при $C_{p(j+1)} = X_{i+1}$ к S_{i+1} добавляем $\langle p, j+1, f, \alpha \rangle$.

Если множество S_{i+1} пусто, то цепочка отвергается.
Если $i = n$ и $S_{i+1} = \{\langle 0, 2, 0, \dashv \rangle\}$, то цепочка допускается.
Конец

Заметим, что упорядочение, вводимое на множествах положений, не имеет смыслового значения, но является просто способом, который позволяет правильно обрабатывать их элементы согласно алгоритму. Заметим также, что i не может стать больше n, если не происходит ни отвержения, ни допускания, потому что $\dashv$ встречается только в 0-м правиле. Но это еще не полное описание алгоритма; надо описать подробно, как эти операции осуществляются на машине. Дальнейшее изложение предполагает знание основ техники обработки списков.

Реализация

(1) Для каждого нетерминала мы храним связанный список его альтернатив, чтобы использовать его в предсказании.

(2) Положения в множестве положений хранятся в связанном списке так, чтобы их можно было обрабатывать по порядку.

(3) Кроме того, когда строится каждое множество положений S_i, мы помещаем записи в вектор длины i. В этом векторе f-я запись ($0 \leqslant f \leqslant i$) есть указатель к списку всех положений в S_i с указателем f, т. е. положений вида $\langle p, j, f, \alpha\rangle \in S_i$ для некоторых p, j, α. Итак, чтобы выяснить, добавлено ли уже положение $\langle p, j, f, \alpha\rangle$ к S_i, мы ищем его в списке, указанном f-й записью в этом векторе. (Время, требуемое на это, не зависит от f.) После построения множества S_i вектор и списки можно отбросить.

(4) Для каждого множества положений S_i и нетерминала N мы храним также список всех положений $\langle p, j, f, \alpha\rangle \in S_i$, для которых $C_{p(j+1)} = N$, чтобы использовать их в восполнителе.

(5) Если грамматика содержит нулевые правила ($A \to \lambda$), мы не можем непосредственно осуществить восполнитель. Когда он осуществляется на нулевом положении ($A \to .\alpha i$), мы хотим добавить к S_i каждое положение из S_i с A справа от точки. Но некоторые из добавленных положений могут привести к добавлению в S_i еще раз. Итак, мы должны выделить этот случай и проверять вновь добавляемые к S_i положения.

Описанная реализация не представляет собой единственного или наилучшего пути осуществления алгоритма. Это просто метод, позволяющий достичь временных и пространственных границ алгоритма, которые мы определим в разд. 5.

5. Границы времени и памяти

Чтобы развить некоторую концепцию эффективности алгоритма, можно рассмотреть формальную модель вычислителя и измерять время числом элементарных шагов, совершаемых этой моделью, а пространство (память) числом использованных единиц памяти. Мы используем модель произвольного доступа к памяти (описанную в дополнении), поскольку полагаем, что эта модель наиболее точно отражает свойства реальных вычислителей, относящиеся к синтаксическому анализу.

Нас интересует зависимость верхних границ времени (и в меньшей степени памяти) от n (длины входной цепочки) для разных классов контекстно-свободных грамматик. Конкретно: n^2-алгоритм для подкласса A грамматик означает, что существует такое число C (которое может зависеть от сложности грамматики, но не от n), что Cn^2 есть верхняя граница числа элементарных шагов, необходимых для анализа любой цепочки длины n относительно грамматики из класса A.

Общий случай. Наш алгоритм для всех контекстно-свободных грамматик является n^3-распознавателем. Доказательство:

(а) Число положений в каждом множестве положений S_i пропорционально i ($\sim i$), поскольку множества значений компонент p, j и α ограничены, а компонента f зависит от i и ограничена числом n.

(б) Считыватель и предсказатель совершают ограниченное число шагов во время обработки одного положения в любом множестве положений. Таким образом, сумма общего времени для обработки положений в S_i и времени на предсказание и считывание пропорциональна i.

(в) Восполнитель совершает в наихудшем случае $\sim i$ шагов на каждое положение, которое он обрабатывает, поскольку он может добавить $\sim f$ положений, полученных из множества положений S_f, определяемых указателем. Таким образом, для обработки всего множества S_i он требует $\sim i^2$ шагов.

(г) Суммируя эти числа шагов по $i = 0, \ldots, n+1$, получаем $\sim n^3$ шагов [1]).

Эта граница сохраняется даже для случая, когда не происходит считывания вперед ($k = 0$). Она не лучше границы Янгера [4] для алгоритма Кока [5], но наш алгоритм лучше по двум причинам. В нашем случае грамматика не обязательно должна быть приведенной к специальному виду (в алгоритме Кока грамматика должна иметь нормальную форму), и, как мы далее покажем, наш алгоритм на большинстве грамматик фактически требует менее n^3 шагов (алгоритм Кока — Янгера всегда требует n^3 шагов). Кроме того, хотя результат Янгера n^3 получен для машин Тьюринга (многоленточных), его алгоритм на машинах с произвольным доступом к памяти не работает быстрее.

Однозначные грамматики. Единственной операцией, заставляющей нас совершать i^2 шагов на обработку каждого множества положений и тем самым дающей в целом время n^3, является восполнитель. Возникает вопрос: в каких случаях эта операция требует только i шагов вместо i^2? После применения восполнителя к множеству положений S_i число положений в нем оказывается самое большее пропорциональным i. Поэтому, если только некоторые из них не были добавлены более одного раза (это может случиться, так что, прежде чем добавлять положение к множеству положений, надо проверять его наличие в этом множестве), на эту операцию потребовалось самое большее $\sim i$ шагов.

Можно показать, что в случае, когда грамматика однозначна и приведена, каждое такое положение добавляется только один раз. В самом деле, предположим, что положение $\langle q, j+1, f, \alpha \rangle$

[1]) Эту оценку, по-видимому, можно улучшить, оценивая более точно суммарное время работы восполнителя. — *Прим. ред.*

при помощи восполнителя добавлено к S_i двумя разными способами. Тогда

$$s_1 = \langle p_1,\ \bar{p}_1,\ f_1,\ X_{i+1} \ldots X_{i+k}\rangle \in S_i,$$
$$s_2 = \langle p_2,\ \bar{p}_2,\ f_2,\ X_{i+1} \ldots X_{i+k}\rangle \in S_i,$$
$$\langle q,\ j,\ f,\ \alpha\rangle \in S_{f_1}, \quad \langle q,\ j,\ f,\ \alpha\rangle \in S_{f_2}, \quad D_{p_1} = C_{q(j+1)} = D_{p_2}$$

и либо $p_1 \neq p_2$, либо $f_1 \neq f_2$, так как в противном случае s_1 и s_2 совпадали бы. Таким образом,

$$X_1 \ldots X_f C_{q1} \ldots C_{q(j+1)} \Rightarrow^* X_1 \ldots X_{f_1} C_{p_1 1} \ldots C_{p_1\bar{p}_1} \Rightarrow^* X_1 \ldots X_i,$$
$$X_1 \ldots X_f C_{q1} \ldots C_{q(j+1)} \Rightarrow^* X_1 \ldots X_{f_2} C_{p_2 1} \ldots C_{p_2\bar{p}_2} \Rightarrow^* X_1 \ldots X_i,$$

а так как равенства $p_1 = p_2$ и $f_1 = f_2$ не могут выполняться одновременно, то два приведенных выше вывода цепочки $X_1 \ldots X_i$ представимы различными деревьями вывода [1]). Учитывая, что грамматика приведена, заключаем отсюда, что каждый нетерминал порождает терминальную цепочку и потому существует неоднозначное предложение $X_1 \ldots X_i\alpha$ для некоторого α.

Итак, в случае однозначной грамматики восполнитель совершает $\sim i$ шагов на обработку каждого множества положений и время ограничено величиной n^2. Заметим, что для грамматик с ограниченной многозначностью время также ограничено величиной n^2, поскольку каждое положение может быть добавлено восполнителем только ограниченное число раз. В работе [1] показано, что граница времени равна n^2 даже для более широкого класса грамматик, и тем самым получен также n^2-результат Янгера [6] для линейных и металинейных грамматик.

Касами и др. [7] независимо получили результат для однозначных грамматик, но алгоритм Касами (который является модификацией алгоритма Кока) требует приведения грамматики к нормальной форме. Его алгоритм, как и наш, достигает своей временно́й границы на машине с произвольным доступом к памяти.

Линейное время. Теперь охарактеризуем класс грамматик, обрабатываемых алгоритмом за время n. Заметим, что для некоторых грамматик число положений в множестве положений может неограниченно возрастать с ростом длины распознаваемой цепочки. Для некоторых других грамматик существует фиксированная граница мощности любого множества

---

[1]) В случае совпадения этих деревьев совпадают также поддеревья, соответствующие выводам $C_{q(j+1)} \Rightarrow X_{f_1} \ldots X_i$ и $C_{q(j+1)} \Rightarrow X_{f_2} \ldots X_i$, откуда следует, что $f_1 = f_2$ и $p_1 = p_2$ (т. е. равны порождаемые цепочки и правила в вершинах поддеревьев). — *Прим. ред.*

положений. Назовем такие грамматики *грамматиками с ограниченными множествами положений* (сокращенно ОМП-грамматиками). Они могут распознаваться алгоритмом за время n. В самом деле, пусть b — граница числа положений в любом множестве положений. Тогда обработка положений в множестве положений вместе со считывателем и предсказателем потребует $\sim b$ шагов, а восполнитель потребует $\sim b^2$ шагов. Суммирование по всем множествам положений дает $\sim b^2 n$, или $\sim n$ шагов.

Таким образом, класс грамматик, обрабатываемых нашим алгоритмом за время n, включает ОМП-грамматики (и не только их). Сравним этот класс с классами грамматик, распознаваемых за линейное время другими алгоритмами. Большинство рассматривавшихся ранее «ограниченных» алгоритмов работало на некоторых подклассах грамматик, которые они могли распознавать за время n, и мы в дальнейшем будем называть их *алгоритмами времени n*.

LR(k)-алгоритм Кнута [2] работает на классе грамматик, который включает почти все другие классы, так что он может служить хорошим эталоном для сравнения. Оказывается, что почти все LR(k)-грамматики (за исключением некоторых праворекурсивных грамматик) являются ОМП-грамматиками. И даже хотя некоторые LR(k)-грамматики могут не быть ОМП-грамматиками, все они могут распознаваться нашим алгоритмом за время n, если используется считывание вперед на k или более символов. На самом деле любое конечное объединение LR(k)-грамматик (полученное непосредственным соединением грамматик для порождения объединения языков) является для нашего алгоритма грамматикой времени n при заданном надлежащем считывании вперед. (Это доказано в работе [1].)

Таким образом, считывание вперед является важной особенностью алгоритма. Границы n^3 и n^2 можно установить и без него, но без него нельзя распознать все LR(k)-грамматики за время n. Кроме того, считывание вперед на $k = 1$ позволяет на практике сократить ненужную работу при обработке многих обычных грамматик. Грамматики времени n для нашего алгоритма включают ОМП-грамматики, конечные объединения LR(k)-грамматик и др. Они включают многие неоднозначные грамматики и даже некоторые с неограниченной степенью неоднозначности, но, к сожалению, существуют также однозначные грамматики, требующие время n^2.

Проиллюстрируем некоторые из идей этого раздела. Грамматика UBDA (табл. 2) требует время, пропорциональное n^3. Заметим, что положение

$$A \rightarrow AA. \dashv x \; 0^2$$

Таблица 2

ГРАММАТИКА UBDA

корень: $A \to x \mid AA$ порождаемый язык: $\{x^n (n \geqslant 1)\}$

REC (UBDA, x^1, 1)

S_0	$\varnothing \to .A \dashv$	$\dashv$		0	S_3	$A \to x.$	$\dashv x$	2
	$A \to .x$	$\dashv x$		0		$A \to AA.$	$\dashv x$	1
	$A \to .AA$	$\dashv x$		0		$A \to AA.$	$\dashv x$	0^2
S_1	$A \to x.$	$\dashv x$		0		$A \to A.A$	$\dashv x$	2
	$\varnothing \to A. \dashv$	$\dashv$		0		$A \to A.A$	$\dashv x$	1
	$A \to A.A$	$\dashv x$		0		$\varnothing \to A. \dashv$	$\dashv$	0
	$A \to .x$	$\dashv x$		1		$A \to A.A$	$\dashv x$	0
	$A \to .AA$	$\dashv x$		1		$A \to .x$	$\dashv x$	3
S_2	$A \to x.$	$\dashv x$		1		$A \to .AA$	$\dashv x$	3
	$A \to AA.$	$\dashv x$		0	S_4	$A \to x.$	$\dashv x$	3
	$A \to A.A$	$\dashv x$		1		$A \to AA.$	$\dashv x$	2
	$\varnothing \to A. \dashv$	$\dashv$		0		$A \to AA.$	$\dashv x$	1^2
	$A \to A.A$	$\dashv x$		0		$A \to AA.$	$\dashv x$	0^3
	$A \to .x$	$\dashv x$		2		$A \to A.A$	$\dashv x$	3
	$A \to .AA$	$\dashv x$		2		$A \to A.A$	$\dashv x$	2
						$A \to A.A$	$\dashv x$	1
						$\varnothing \to A. \dashv$	$\dashv$	0
						$A \to A.A$	$\dashv x$	0
						$A \to .x$	$\dashv x$	4
						$A \to .AA$	$\dashv x$	4
					S_5	$\varnothing \to A \dashv .$	$\dashv$	0

дважды добавляется восполнителем. Это обозначено верхним индексом при 0. Рассматривая верхние индексы в положениях

$$A \to AA. \quad \dashv x \quad 2$$
$$A \to AA. \quad \dashv x \quad 1^2$$
$$A \to AA. \quad \dashv x \quad 0^3$$

в S_4, можно сказать, что в каждом из $\sim n$ множеств положений существует $\sim i$ положений, которые добавляются $\sim i$ раз; в результате получается n^3 действий.

Грамматика ВК (табл. 3) имеет неограниченную неоднозначность, но является грамматикой времени n и фактически ОМП-грамматикой. Это можно усмотреть из того, что все множества положений после S_1 имеют одинаковую мощность. Грамматика PAL (табл. 4) однозначна и требует время n^2. Множе-

ства S_i, S_{i+1} и т. д. до S_n включительно содержат каждое по $i+4$ положений, так что общее число положений оказывается пропорциональным n^2.

Таблица 3

ГРАММАТИКА BK

корень: $K \to \mid KJ$ порождаемый язык: $\{x^n\ (n \geqslant 0)\}$
$J \to F \mid I$
$F \to x$
$I \to x$

REC (BK, x^n, 1)

S_0	$\varnothing \to .K \dashv$	$\dashv$		0	S_i	$(2 \leqslant i \leqslant n)$		
	$K \to .$	$\dashv$	x	0		$F \to x.$	$\dashv x$	$i-1$
	$K \to .KJ$	$\dashv$	x	0		$F \to x.$	$\dashv x$	$i-1$
	$\varnothing \to K.\dashv$	$\dashv$		0		$J \to F.$	$\dashv x$	$i-1$
	$K \to K.J$	$\dashv$	x	0		$J \to I.$	$\dashv x$	$i-1$
	$J \to .F$	$\dashv$	x	0		$K \to KJ.$	$\dashv x$	0^2
	$J \to .I$	$\dashv$	x	0		$K \to K.J$	$\dashv x$	0
	$F \to .x$	$\dashv$	x	0		$\varnothing \to K. \dashv$		0
	$I \to .x$	$\dashv$	x	0		$J \to .F$	$\dashv x$	i
S_1	$F \to x.$	$\dashv$	x	0		$J \to .I$	$\dashv x$	i
	$I \to x.$	$\dashv$	x	0		$F \to .x$	$\dashv x$	i
	$J \to F.$	$\dashv$	x	0		$I \to .x$	$\dashv x$	i
	$J \to I.$	$\dashv$	x	0	S_{n+1}			
	$K \to KJ.$	$\dashv$	x	0^2		$\varnothing \to K \dashv .$	$\dashv$	**0**
	$K \to K.J$	$\dashv$	x	0				
	$\varnothing \to K. \dashv$	$\dashv$		0				
	$J \to .F$	$\dashv$	x	1				
	$J \to .I.$	$\dashv$	x	1				
	$F \to .x$	$\dashv$	x	1				
	$I \to .x$	$\dashv$	x	1				

Память. Поскольку в памяти находится $\sim n$ множеств положений, в каждом по $\sim n$ положений, память ограничена в общем случае числом n^2. Ту же границу дают алгоритмы Кока и Касами, но в нашем алгоритме n^2 — это лишь верхняя граница, а алгоритм Кока всегда требует n^2 времени.

6. Эмпирические результаты

Мы запрограммировали свой алгоритм и сравнили его с исходящим и восходящим анализами, проведенными по Гриффитсу и Петрику [8]. Это старейшие из алгоритмов анализа контекстно-

Таблица 4

ГРАММАТИКА PAL

$A \to x \mid xAx$ порождаемый язык: $\{x^n (n \geqslant 1,\ n$ нечетное$\}$

REC (PAL, x^5, 0)

S_0	$\varnothing \to .A \dashv$	0	S_4	$A \to xAx.$	1
	$A \to .x$	0		$A \to x.$	3
	$A \to .xAx$	0		$A \to x.Ax$	3
S_1	$A \to x.$	0		$A \to xA.x$	0
	$A \to x.Ax$	0		$A \to xA.x$	2
	$\varnothing \to A. \dashv$	0		$A \to .x$	4
	$A \to .x$	1		$A \to .xAx$	4
	$A \to .xAx$	1	S_5	$A \to xAx.$	0
S_2	$A \to x.$	1		$A \to xAx.$	2
	$A \to x.Ax$	1		$A \to x.$	4
	$A \to xA.x$	0		$A \to x.Ax$	4
	$A \to .x$	2		$\varnothing \to A. \dashv$	0
	$A \to .xAx$	2		$A \to xA.x$	1
S_3	$A \to xAx.$	0		$A \to xA.x$	3
	$A \to x.$	2		$A \to .x$	5
	$A \to x.Ax$	2		$A \to xAx.$	5
	$\varnothing \to A. \dashv$	0	S_6	$\varnothing \to A \dashv .$	0
	$A \to xA.x$	1			
	$A \to .x$	3			
	$A \to .xAx$	3			

свободных грамматик, и они сильно зависят от попятных движений. Их верхние границы для времени, быть может по этой причине, являются экспоненциальными (C^n для некоторой константы C). Однако на некоторых грамматиках эти алгоритмы могут работать хорошо, и оба были применены в различных системах построения трансляторов, так что их будет интересно сравнить с нашим алгоритмом.

Алгоритмы Гриффитса и Петрика описаны в терминах не реальных единиц времени, а «примитивных операций». Эти алгоритмы представлены как множества недетерминированных правил подстановок для устройств, подобных машине Тьюринга. Каждое применение одного из правил есть примитивная операция. Мы в качестве примитивной операции выбрали добавление положения к множеству положений (или попытку добавления положения, когда оно уже содержалось в множестве). Мы

считаем, что это действие[1]) сравнимо с их примитивной операцией, поскольку оба они являются в некотором смысле наиболее сложными операциями, совершаемыми алгоритмом, сложность которых не зависит ни от объема грамматики, ни от входной цепочки.

Мы сравним эти алгоритмы на семи различных грамматиках. Два из них не были еще использованы, поскольку не были заданы точные грамматики. В первых четырех примерах Гриффитс и Петрик нашли окончательные выражения для своих результатов, и мы также сделали это (см. табл. 5; BU и TD — восходящий и нисходящий алгоритмы соответственно; SBU и STD — их улучшенные варианты). Из приведенных результатов видно, что SBU намного лучше других алгоритмов; остальные данные подтверждают это. Поэтому мы сравним наш алгоритм только с SBU. Наш алгоритм будет применен при $k = 0$. Алгоритмы сравнимы на простых грамматиках $G1$, $G2$, $G3$, но на грамматике $G4$, являющейся очень неоднозначной, превосходство нашего алгоритма очевидно: от n к n^3.

Таблица 5

$G1$		$G2$		$G3$	$G4$	
корень: $S \to Ab$, $A \to a \mid Ab$		корень: $S \to aB$, $B \to aB \mid b$		корень: $S \to ab \mid aSb$	корень: $S \to AB$, $A \to a \mid Ab$, $B \to bc \mid bB \mid Bd$	
Грамматика	Порождаемый язык	TD	STD	BU	SBU	Наш метод
$G1$	ab^n	$(n^2+7n+2)/2$	$(n^2+7n+2)/2$	$9n+5$	$9n+5$	$4n+7$
$G2$	a^nb	$3n+2$	$2n+2$	$11 \cdot 2^n + 7$	$4n+4$	$4n+4$
$G3$	a^nb^n	$5n-1$	$5n-1$	$11 \cdot 2^{n-1} - 5$	$6n$	$6n+4$
$G4$	ab^ncd	$\sim 2^{n+6}$	$\sim 2^{n+2}$	$\sim 2^{n+5}$	$(n^3+21n^2+46n+15)/3$	$18n+8$

Для следующих трех грамматик мы представим лишь необработанные данные (табл. 6—8). Данные для нашего алгоритма были получены при программировании его вместе с программой для вычисления числа совершаемых им примитивных операций. Мы включили также данные из работы [8] на предсказывающем анализаторе PA, представляющем собой модификацию нисходящего алгоритма. На грамматике для исчисления высказываний

[1]) Заметим, что использованная автором примитивная операция при реализации на машине Тьюринга потребует порядка $\log n$ шагов. — *Прим. ред.*

Таблица 6

Грамматика для исчисления высказываний

корень: $F \to C \mid S \mid P \mid U$
$C \to U \supset U$
$U \to (F) \mid \sim U \mid L$
$L \to L' \mid p \mid q \mid r$
$S \to U \vee S \mid U \vee U$
$P \to U \wedge P \mid U \wedge U$

Порождаемый язык	Длина	PA	SBU	Наш метод
p	1	14	18	28
$(p \wedge q)$	5	86	56	68
$(p' \wedge q) \vee r \vee p \vee q'$	13	232	185	148
$p \supset ((q \supset \sim (r' \vee (p \wedge q)))$ $\supset (q' \vee r)$	26	712	277	277
$\sim (\sim p' \wedge (q \vee r) \wedge p'))$	17	1955	223	141
$((p \wedge q) \vee (q \wedge r) \vee (r \wedge p'))$ $\supset \sim ((p' \vee q') \wedge (r' \vee p))$	38	2040	562	399

Таблица 7

ГРАММАТИКА GRE

корень: $X \to a \mid Xb \mid Ya$
$Y \to e \mid XdY$

Порождаемый язык	Длина	PA	SBU	Наш метод
$ededea$	6	35	52	33
$ededeab^1$	10	75	92	45
$ededeab^{10}$	16	99	152	63
$ededeab^{200}$	206	859	2052	633
$(ed)^4eabb$	12	617	526	79
$(ed)^7eabb$	18	24 352	16 336	194
$(ed)^8eabb$	20	86 139	54 660	251

PA, по-видимому, работает за время n^2, тогда как SBU и наш алгоритм работают каждый за время n, причем наш немного быстрее. Грамматика GRE дает два вида поведения. Все три алгоритма линейно растут с ростом числа символов b, при этом в SBU получается сравнительно большой постоянный коэф-

Таблица 8

ГРАММАТИКА NSE

корень: $S \to AB$
$A \to a \mid SC$
$B \to b \mid DB$
$C \to c$
$D \to d$

Порождаемый язык	Длина	SBU	Наш метод
$adbcddb$	7	43	44
$ad^3bcbcd^3bcd^1b$	18	111	108
$adbcd^2bcd^5bcd^3b$	19	117	114
$ad^{18}b$	20	120	123
$a\,(bc)^3\,d^3\,(bcd)^2\,dbcd^1b$	24	150	141
$a\,(bcd)^2\,dbcd^3bcb$	16	100	95

фициент. Однако PA и SBU растут экспоненциально с ростом числа подцепочек *ed*, а наш алгоритм растет квадратично. Грамматика NSE совсем простая, и каждый алгоритм требует время n с одним и тем же коэффициентом.

Итак, ясно, что наш алгоритм лучше алгоритмов с попятным движением. На каждой из семи рассмотренных грамматик он работает так же, как лучший из них, а на некоторых — даже значительно лучше.

Известны по крайней мере четыре различных общих алгоритма распознавания контекстно-свободных грамматик, кроме нашего: TD, BU, Касами n^2 и Кока n^3. Мы уже показали, что граница времени у нашего алгоритма не хуже, чем у любого из этих алгоритмов. Интересно сравнить наш алгоритм с этими алгоритмами в практическом отношении, а не только по верхним границам.

Мы только что представили некоторые эмпирические результаты, показывающие, что наш алгоритм лучше, чем TD и BU. Кроме того, наш алгоритм лучше алгоритма Кока, поскольку последний всегда достигает своей верхней границы n^3. Это верно и для алгоритма Касами. Его алгоритм [7] описан как алгоритм для однозначных грамматик, но его легко расширить до общего алгоритма. Мы полагаем, что его временна́я граница равна n^3 для общего случая и равна n^2 так же часто, как и у нашего алгоритма. Результаты о классе грамматик, анализируемых алгоритмом Касами за время n, нам неизвестны.

7. Практическое использование алгоритма

В этом разделе мы обсудим вопрос, в каких областях и в какой форме лучше всего применять наш алгоритм.

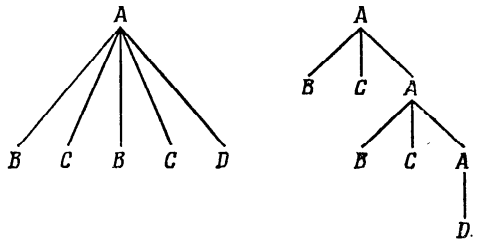
Форма. Прежде чем что-то делать в этом направлении, надо преобразовать распознаватель в анализатор. Это осуществляется таким изменением распознавателя, чтобы он в процессе распознавания строил дерево анализа. Каждый раз, когда восполнитель добавляет положение $E \rightarrow \alpha D.\beta g$ (игнорируя считывание вперед), строится указатель от вхождения D в это положение к положению $D \rightarrow \gamma.f$, которое вынудило нас сделать эту операцию. Это показывает, что γ было проанализировано как порождение нетерминала D. В случае неоднозначности из D будет построено несколько указателей, по одному для каждого восполнителя, заставляющего добавить к данному множеству положений положение $E \rightarrow \alpha D.\beta g$. У каждого символа в γ (если это не терминал) также будут исходящие из него указатели и т. д. Все вместе представляет дерево вывода для D. Таким образом, когда мы достигаем заключительного положения $\varnothing \rightarrow R \dashv .0$, у нас уже есть дерево анализа для предложения, «подвешенного» к R, если это предложение однозначно. В противном случае у нас будет «склеенное» представление всех возможных деревьев анализа. В [1] дано точное описание этого процесса.

Временны́е границы для анализатора те же, что и для распознавателя, тогда как граница памяти возрастает в общем случае до n^3, чтобы можно было запоминать деревья анализа.

Мы рекомендуем все время использовать считывание вперед с $k = 1$. Осуществление считывания вперед для любого другого k стоило бы дороже при программировании и вообще было бы менее эффективно; очень мало языков программирования нуждаются в большом считывании вперед. Бо́льшая часть языков программирования для снятия неоднозначности их конструкций использует только однобуквенный контекст, и если в некоторых случаях две буквы окажутся необходимыми, то наш алгоритм обладает тем хорошим свойством, что это не помешает его применению, а может лишь несколько удлинить его работу.

Наш алгоритм обладает тем полезным свойством, что его можно приспособить для обращения с расширениями контекстно-свободных грамматик, в котором используется обозначение операции «звезда» Клини: $A \rightarrow \{BC\}^* D$ означает, что A можно заменить произвольным числом (включая 0) цепочек BC, за которыми следует D. Это правило порождает язык, эквивалентный языку, порождаемому грамматикой $A \rightarrow D \mid BCA$. Однако

структуры анализа языка в этих грамматиках различны:



Структуры, подобные структуре слева, нельзя получить в контекстно-свободной грамматике, так что это расширение оказывается полезным. Модификация нашего алгоритма, осуществляющая этот анализ, включает две дополнительные операции:

(1) Любое положение вида

$$A \rightarrow \alpha . \{\beta\}^* \gamma \; f$$

заменяется положениями

$$A \rightarrow \alpha \{. \beta\}^* \gamma \; f$$
$$A \rightarrow \alpha \{\beta\}_* . \gamma \; f$$

указывающими, может ли β присутствовать.

(2) Любое положение вида

$$A \rightarrow \alpha \{\beta .\}^* \gamma \; f$$

заменяется положениями

$$A \rightarrow \alpha \{. \beta\}^* \gamma \; f$$
$$A \rightarrow \alpha \{\beta\}^* . \gamma \; f$$

указывающими, может ли β повторяться.

Использование. Наиболее полезным наш алгоритм, вероятно, будет в системах обработки естественных языков, в которых в полную силу применяются контекстно-свободные грамматики. Его можно использовать также в компиляторах и расширяющихся языках. В большинстве компиляторов и расширяющихся языков программист может выразить синтаксис (или синтаксическое расширение) своего языка способом, подобным языку Бэкуса, после чего для разбора программ в этом языке система применит анализатор. Грамматики языков программирования обычно принадлежат ограниченному подмножеству контекстно-свободных грамматик, которые могут анализироваться эффективно, однако некоторые системы компиляции в действительности используют общие анализаторы, и в этом случае

можно использовать наш алгоритм. В дополнение к его эффективности он допускает грамматики в той форме, в которой они записаны, так что семантические подпрограммы можно ассоциировать с правилами и не опасаться, что анализатор не отражает исходную структуру грамматики.

Однако по сравнению с алгоритмами времени n наш алгоритм не столь хорош. Хотя он анализирует за время n любую грамматику, которую может обработать анализатор времени n, в этом сравнении не принимается в расчет величина постоянного коэффициента при n. Большинство алгоритмов времени n в действительности состоит из двукратного процесса. Сначала они из допустимой грамматики составляют для нее анализатор, а затем эту грамматику отбрасывают и построенный анализатор применяют непосредственно к анализируемым цепочкам. Это позволяет алгоритмам времени n включить в построенный анализатор значительную часть специализированной информации, сводя тем самым коэффициент при n к некоторой довольно малой величине — вероятно, меньшего порядка, чем коэффициент в нашем алгоритме.

В связи с этим мы развили для нашего алгоритма процесс компиляции, который работает только в границах времени n и приводит наш коэффициент к величине приблизительно такого же порядка, как у анализаторов времени n. Это может сделать наш алгоритм способным конкурировать с ними, но мы не осуществили реализации и испытаний, так что это пока лишь предположение. Необходим, однако, некоторый вид эффективных анализаторов времени n для широкого класса грамматик, поскольку наиболее ограниченные анализаторы обладают тем недостатком, что для многих языков программирования написанная естественным образом грамматика не приемлема для анализа и, чтобы сделать ее приемлемой, над ней надо произвести много специфических преобразований. Алгоритм Кнута представляет исключение, но с ним связана проблема слишком большого объема компилятора для грамматик применяемых языков программирования (см. [1]). К сожалению, построенный нами алгоритм, будучи похожим на алгоритм Кнута, сталкивается с теми же трудностями.

8. Заключение

Подчеркнем еще раз, что наш алгоритм не только согласуется с лучшими прежними результатами для времени (n^3 (Янгер), n^2 (Касами) и n (Кнут)) и даже превосходит их, но эти результаты дает единственный алгоритм, не приспосабливаемый к тому классу грамматик, над которыми он работает, и не требующий приведения грамматики в специальную форму. Дру-

гими словами, в то время как алгоритм Кнута работает только на LR(k)-грамматиках, а алгоритм Касами (по крайней мере в том виде, в котором он изложен в его статье) — только на однозначных грамматиках, наш алгоритм работает на всех этих грамматиках и делает это, по-видимому, так же хорошо, как другие алгоритмы.

Дополнение

Машины с произвольным доступом к памяти. В этой модели имеется неограниченное число регистров (счетчиков), каждый из которых может содержать любое неотрицательное целое число. Они именуются (адресованы) последовательными неотрицательными целыми числами. На этих регистрах производятся следующие примитивные операции:

(1) Поместить 0 или содержимое одного регистра в другой.
(2) Проверить равенство нулю содержимого одного регистра или равенство содержимых двух регистров.
(3) Прибавить 1 или вычесть 1 из содержимого регистра (полагая $0 - 1 = 0$).
(4) Прибавить содержимое одного регистра к другому.

Управление в этой модели осуществляется обычным устройством с конечным числом состояний (конечным автоматом). Самое важное свойство этой машины состоит в том, что в четырех описанных выше операциях регистр R можно указать двумя способами:

(1) R — регистр с адресом n (регистр n).
(2) R — регистр, адресом которого является содержимое регистра n.

Второй способ (иногда называемый непрямой адресацией) вместе с примитивной операцией (4), применяемой для возможности доступа к любому месту в схеме, наделяет нашу модель свойством произвольного доступа к памяти. Время измеряется числом совершаемых примитивных операций, а память (пространство) — числом использованных регистров.

Список литературы

1. Earley J., An efficient context-free parsing algorithm, Ph. D. Thesis, Comput. Sci. Dept; Carnegie-Mellon U., Pittsburgh, Pa., 1968.
2. Knuth D. E., On the translation of languages from left to right, *Inform. and Control*, 8 (1965), 607—639. (Русский перевод см. в настоящем сборнике, стр. 9—42.)
3. Floyd R. W., The syntax of programming languages — a survey, *IEEE Trans*, **EC-13**, 4 (August 1964).
4. Younger D. H., Recognition and parsing of context-free languages in time n^3, *Inform. and Control*, **10** (1967), 189—208. (Русский перевод: Янгер Д.,

Распознавание и анализ контекстно-свободных языков за время n^3, в сб. «Проблемы математической логики», изд-во «Мир», М., 1970, стр. 344—362.)
5. Hays D., Automatic language-data processing, в сб. «Computer Applications in the Behavioral Sciences» под ред. Borko H., Prentice Hall, Englewood Cliffs, N. J., 1962.
6. Younger D. H., Context-free language processing in time n^3, General Electric R&D Center, Schenectady, N. Y., 1966.
7. Kasami T., Torii K., A syntax-analysis procedure for unambiguous context-free grammars, *J. ACM*, **16**, 3 (1969), 423—431.
8. Griffiths T., Petrick S., On the relative efficiencies of context-free grammar recognizers, *Comm. ACM*, **8**, 5 (1965), 289—300.
9. Feldman J., Gries D., Translator writing systems, *Comm. ACM*, **11**, 2 (1968), 77—113.