Волкова И.А., Руденко Т.В.

Системное программное обеспечение

Формальные грамматики и языки. Элементы теории трансляции.

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, некоторые из них могут объединяться в одну фазу, другие могут выполнятся в течение всего процесса компиляции. В интерпретаторах и при смешанной стратегии трансляции некоторые этапы могут вообще отсутствовать.

В этом 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). Все алгоритмы записаны на Си.

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

Описание модельного языка

P → **program** D1; B

D1 → **var** D {;D}

D → I {,I}: [**int** | **bool**]

B → **begin** S {;S} **end**

S → I := E | **if** E **then** S **else** S | **while** E **do** S | B | **read** (I) | **write** (E)

E → E1 [= | < | > | !=] E1

E1 → T {[+ | - | *or*] T}

T → F {[* | / | *and*] F}

F → I | N | L | *not* F | (E)

L → **true** | **false**

I → C | IC | IR

N → R | NR

C → *a* | *b* | ... | *z* | *A* | *B* | ... |*Z*

R → *0* | *1* | *2* | ... | *9*

Замечание:

1. 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.
2. запись вида [α | β] означает, что в порождаемой цепочке в этом месте может находиться либо α, либо β.
3. 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 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.

Лексический анализ

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 грамматику.

Напомним, что грамматика G = (VT, VN, P, S) называется *леволинейной*, если каждое правило из P имеет вид A → Bt либо A → t, где A ⊂ VN, B ⊂ VN, t ⊂ VT.

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

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 $A \to a_1$ (другими словами, производим "свертку" терминала a_1 к нетерминалу A)

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

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

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 $a_1a_2...a_n\perp \subset L(G)$.

(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 $a_1a_2...a_n\perp \subseteq L(G)$.

(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 $B \to Aa_i$. Это означает, что исходная цепочка $a_1a_2...a_n\perp \subseteq L(G)$.

(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 грамматики $G = (\{a, b, \perp\}, \{S, A, B, C\}, P, S)$, такая таблица будет выглядеть следующим образом:

P: $S \to C\perp$

$C \to Ab \mid Ba$

A → a | Ca

B → b | Cb

	a	b	□
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:

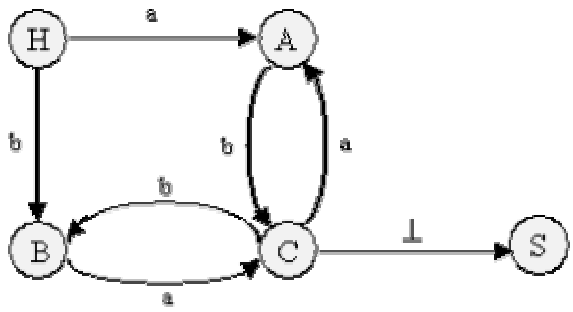
(1) строят вершины графа, помеченные нетерминалами грамматики (для каждого нетерминала - одну вершину), и еще одну вершину, помеченную символом, отличным от нетерминальных (например, H). Эти вершины будем называть *состояниями***.** H - начальное состояние.

(2) соединяем эти состояния дугами по следующим правилам:

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

б) для каждого правила W → Vt соединяем дугой состояния V и W (от V к W) и помечаем дугу символом t;

Диаграмма состояний для грамматики G (см. пример выше):



Алгоритм разбора по диаграмме состояний:

(1) объявляем текущим 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 конечное (если автомат завершает работу переходом в это состояние, то анализируемая цепочка им допускается).
Определение: *конечный автомат (КА)* - это пятерка (K, VT, F, H, S), где
K - конечное множество состояний;
VT - конечное множество допустимых входных символов;
F - отображение множества $K \times VT \to K$, определяющее поведение автомата; отображение F часто называют функцией переходов;
$H \subset K$ - начальное состояние;
$S \subset K$ - заключительное состояние (либо конечное множество заключительных состояний).
F(A, t) = B означает, что из состояния A по входному символу t происходит переход в состояние B.
Определение: конечный автомат *допускает цепочку* $a_1a_2...a_n$, если $F(H,a_1) = A_1$; $F(A_1,a_2) = A_2$; . . . ; $F(A_{n-2},a_{n-1}) = A_{n-1}$; $F(A_{n-1},a_n) = S$, где $a_i \subset VT$, $A_j \subset K$, $j = 1, 2, ..., n-1$; $i = 1, 2, ..., n$; H - начальное состояние, S - одно из заключительных состояний.
Определение: множество цепочек, допускаемых конечным автоматом, составляет определяемый им *язык*.
Для более удобной работы с диаграммами состояний введем несколько соглашений:

1. если из одного состояния в другое выходит несколько дуг, помеченных разными символами, то будем изображать одну дугу, помеченную всеми этими символами;

2. непомеченная дуга будет соответствовать переходу при любом символе, кроме тех, которыми помечены другие дуги, выходящие из этого состояния.
3. 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 = ({a,b, ⊥}, {S,A,B,C}, P, S), где

P: S → C⊥

C → Ab | Ba

A → a | Ca

B → b | Cb

анализатор будет таким:

```
#include <stdio.h>
int scan_G(){
 enum state {H, A, B, C, S, ER};  /* множество состояний */
 state CS;  /* CS - текущее состояние */
 FILE *fp; /* указатель на файл, в котором находится анализируемая цепочка */
 int c;

 CS=H;
 fp = fopen ("data","r");
 c = fgetc (fp);
 do {switch (CS) {
   case H: if (c == 'a') {c = fgetc(fp); CS = A;}
         else if (c == 'b') {c = fgetc(fp); CS = B;}
            else CS = ER;
     break;
   case A: if (c == 'b') {c = fgetc(fp); CS = C;}
         else CS = ER;
         break;
   case B: if (c == 'a') {c = fgetc(fp); CS = C;}
         else CS = ER;
        break;
   case C: if (c =='a') {c = fgetc(fp); CS = A;}
         else if (c == 'b') {c = fgetc(fp); CS = B;}
            else if (c == '⊥') CS = S;
          else CS = ER;
     break;
          }
   } while (CS != S && CS != ER);
```

```
 if (CS == ER) return -1; else return 0;
}
```

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 = ({a,b, ⊥}, {S,A,B}, P, S), 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 (K, VT, F, H, S), где

K - конечное множество состояний;

VT - конечное множество допустимых входных символов;

F - отображение множества K × VT в множество подмножеств K;

H K - конечное множество начальных состояний;

S K - конечное множество заключительных состояний.

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

В этом случае можно предложить алгоритм, который будет перебирать все возможные варианты сверток (переходов) один за другим; если цепочка принадлежит языку, то будет найден путь, ведущий к успеху; если будут просмотрены все варианты, и каждый из них будет завершаться 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.

Это означает, что для любого НКА всегда можно построить детерминированный КА, определяющий тот же язык.

Алгоритм построения детерминированного КА по НКА

Вход: M = (K, VT, F, H, S) - недетерминированный конечный автомат.

Выход: M’ = (K’, VT, F’, H’, S’) - детерминированный конечный автомат, допускающий тот же язык, что и автомат M.

Метод:

1. Множество состояний K’ состоит из всех подмножеств множества K. Каждое состояние из K’ будем обозначать $[A_1A_2...A_n]$, где A_i □ K.
2. Отображение F’ определим как F’ $([A_1A_2...A_n], t) = [B_1B_2...B_m]$, где для каждого $1 \le j \le m$ $F(A_i, t) = B_j$ для каких-либо $1 \le i \le n$.
3. Пусть $H = \{H_1, H_2, ..., H_k\}$, тогда $H' = [H_1, H_2, ..., H_k]$.
4. Пусть $S = \{S_1, S_2, ..., S_p\}$, тогда S’ - все состояния из K’, имеющие вид $[...S_i...]$, $.S_i$ □ S для какого-либо $1 \le i \le p$.

Замечание: в множестве K’ могут оказаться состояния, которые недостижимы из начального состояния, их можно исключить.

Проиллюстрируем работу алгоритма на примере.

Пусть задан НКА M = ({H, A, B, S}, {0, 1}, F, {H}, {S}), где

F(H, 1) = B F(B, 0) = A
F(A, 1) = B F(A, 1) = S ,

тогда соответствующий детерминированный конечный автомат будет таким:

K’ = {[H], [A], [B], [S], [HA], [HB], [HS], [AB], [AS], [BS], [HAB], [HAS], [ABS], [HBS], [HABS]}

F’([A], 1) = [BS] F’([H], 1) = [B]
F’([B], 0) = [A] F’([HA], 1) = [BS]
F’([HB], 1) = [B] F’([HB], 0) = [A]
F’([HS], 1) = [B] F’([AB], 1) = [BS]
F’([AB], 0) = [A] F’([AS], 1) = [BS]
F’([BS], 0) = [A] F’([HAB], 0) = [A]
F’([HAB], 1) = [BS] F’([HAS], 1) = [BS]
F’([ABS], 1) = [BS] F’([ABS], 0) = [A]
F’([HBS], 1) = [B] F’([HBS], 0) = [A]
F’([HABS], 1) = [BS] F’([HABS], 0) = [A]

S’ = {[S], [HS], [AS], [BS], [HAS], [ABS], [HBS], [HABS]}

Достижимыми состояниями в получившемся КА являются [H], [B], [A] и [BS], поэтому остальные состояния удаляются.

Таким образом, M’ = ({[H], [B], [A], [BS]}, {0, 1}, F’, H, {[BS]}), где

F’([A], 1) = [BS] F’([H], 1) = [B]
F’([B], 0) = [A] F’([BS], 0) = [A]

Задачи лексического анализа

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

Лексический анализ важен для процесса компиляции по нескольким причинам:

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

b) лексический анализ уменьшает длину программы, устраняя из ее исходного представления несущественные пробелы и комментарии;

c) 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.

Например, идентификатор (I):

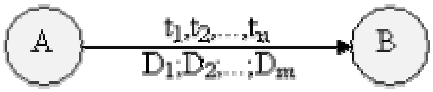
I → a|b|...|z|Ia|Ib|...|Iz|I0|I1|...|I9

целое без знака (N):

N→ 0|1|...|9|N0|N1|...|N9

и т.д.

Для грамматик этого класса, как мы уже видели, существует простой и эффективный алгоритм анализа того, принадлежит ли заданная цепочка языку, порождаемому этой грамматикой. Однако перед лексическим анализатором стоит более сложная задача: он должен сам выделить в исходном тексте цепочку символов, представляющую лексему, а также преобразовать ее в пару (тип_лексемы, указатель_на_информацию_о_ней). 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 ,... n, то осуществляется переход в состояние B; при этом необходимо выполнить действия D_1, D_2, ... ,D_m.

Лексический анализатор для М-языка

Вход лексического анализатора - символы исходной программы на М-языке; результат работы - исходная программа в виде последовательности лексем (их внутреннего представления).

Лексический анализатор для модельного языка будем писать в два этапа: сначала построим диаграмму состояний с действиями для распознавания и формирования внутреннего представления лексем, а затем по ней напишем программу анализатора.

Первый этап: разработка ДС.

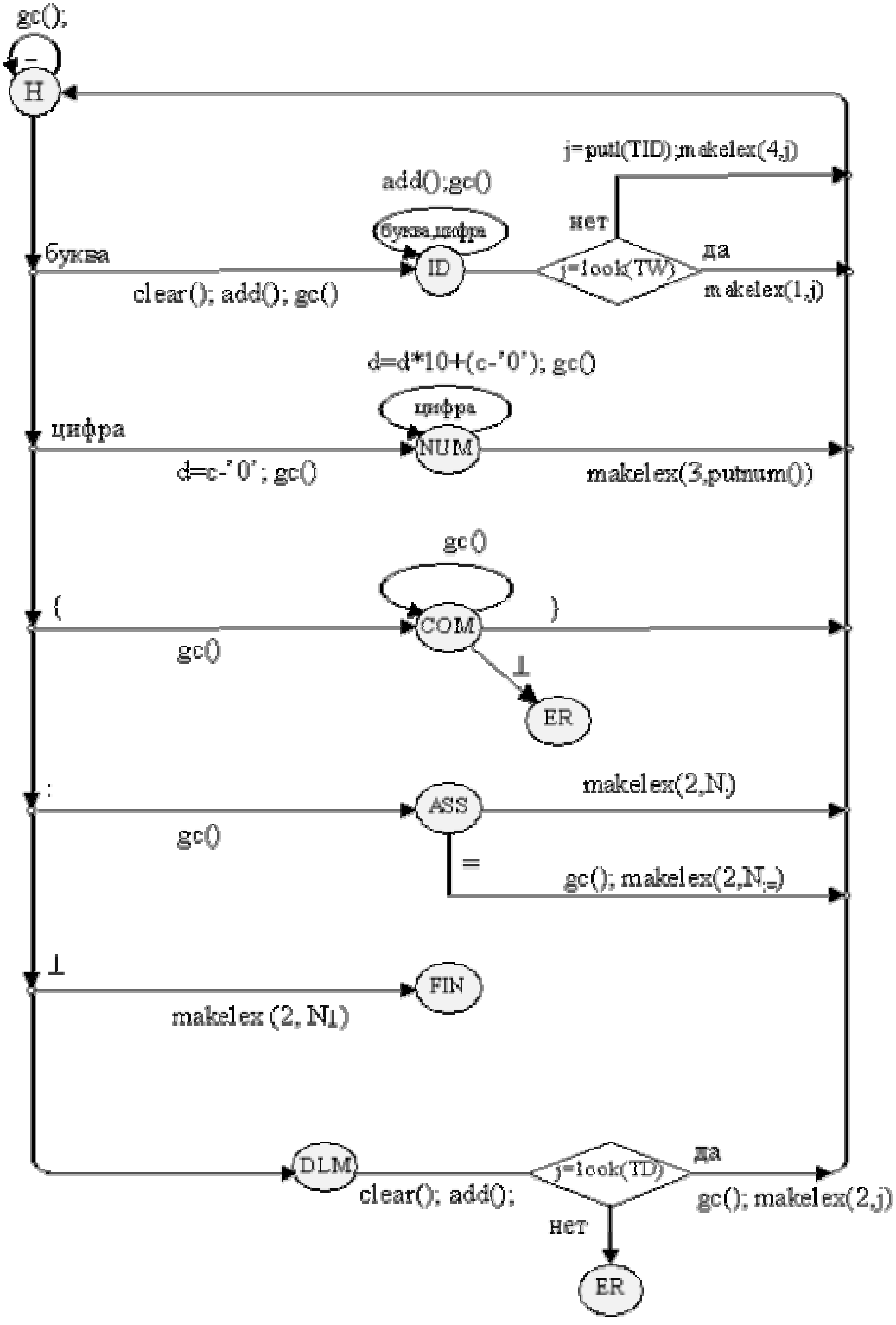
Представление лексем: все лексемы М-языка разделим на несколько классов; классы перенумеруем:

- служебные слова - 1,
- ограничители - 2,
- константы (целые числа) - 3,
- идентификаторы - 4.

Внутреннее представление лексем - это пара (номер_класса, номер_в_классе). Номер_в_классе - это номер строки в таблице лексем соответствующего класса.
Соглашение об используемых переменных, типах и функциях:
1) пусть есть переменные:
buf - буфер для накопления символов лексемы;
c - очередной входной символ;
d - переменная для формирования числового значения константы;
TW - таблица служебных слов M-языка;
TD - таблица ограничителей M-языка;
TID - таблица идентификаторов анализируемой программы;
TNUM - таблица чисел-констант, используемых в программе.
Таблицы TW и TD заполнены заранее, т.к. их содержимое не зависит от исходной программы; TID и TNUM будут формироваться в процессе анализа; для простоты будем считать, что все таблицы одного типа; пусть tabl - имя типа этих таблиц, ptabl - указатель на tabl.

1. пусть есть функции:

void clear (void); - очистка буфера buf;
void add (void); - добавление символа c в конец буфера buf;
int look (ptabl T); - поиск в таблице T лексемы из буфера buf; результат: номер строки таблицы с информацией о лексеме либо 0, если такой лексемы в таблице T нет;
int putl (ptabl T); - запись в таблицу T лексемы из буфера buf, если ее там не было; результат: номер строки таблицы с информацией о лексеме;
int putnum (); - запись в TNUM константы из d, если ее там не было; результат: номер строки таблицы TNUM с информацией о константе-лексеме;
void makelex (int k, int i); - формирование и вывод внутреннего представления лексемы; k - номер класса, i - номер в классе;
void gc (void) - функция, читающая из входного потока очередной символ исходной программы и заносящая его в переменную c.
Тогда диаграмма состояний для лексического анализатора:



Замечание: символом N_x в диаграмме (и в тексте программы) обозначен номер лексемы x в ее классе.

Второй этап: по ДС пишем программу

```
#include <stdio.h>
#include <ctype.h>
#define BUFSIZE 80
typedef struct tabl * ptabl;
extern ptabl TW, TID, TD, TNUM;
char buf[BUFSIZE]; /* для накопления символов лексемы */
```

```
int c;   /* очередной символ */
int d;   /* для формирования числового значения
 константы */
void clear(void); /* очистка буфера buf */
void add(void); /* добавление символа c в конец буфера
          buf*/
int look(ptabl);  /* поиск в таблице лексемы из buf;
          результат: номер строки таблицы либо 0 */
int putl(ptabl);  /* запись в таблицу лексемы из buf,
          если ее там не было; результат:
 номер строки таблицы */
int putnum();    /* запись в TNUM константы из d, если ее
          там не было; результат: номер строки
          таблицы TNUM */
int j;      /* номер строки в таблице, где находится
          лексема, найденная функцией look */
void makelex(int,int); /* формирование и вывод внутреннего
 представления лексемы */
void scan (void)
 {enum state {H,ID,NUM,COM,ASS,DLM,ER,FIN};
 state TC; /* текущее состояние */
 FILE* fp;
  TC = H;
  fp = fopen("prog","r"); /* в файле "prog" находится
 текст исходной программы */
 c = fgetc(fp);
  do {switch (TC) {
   case H:
    if (c == ' ') c = fgetc(fp);
 else if (isalpha(c))
       {clear(); add(); c = fgetc(fp); TC = ID;}
        else if (isdigit (c))
    {d = c - '0'; c = fgetc(fp); TC = NUM;}
            else if (c=='{') {c=fgetc(fp); TC = COM;}
             else if (c == ':')
  {c = fgetc(fp); TC = ASS;}
             else if (c == '⊥')
 {makelex(2, N⊥); TC = FIN;}
         else TC = DLM;
    break;
   case ID:
```

```
   if (isalpha(c) || isdigit(c)) {add(); c=fgetc(fp);}
   else {if (j = look (TW)) makelex (1,j);
        else {j = putl (TID); makelex (4,j);};
   TC = H;};
   break;
  case NUM:
   if (isdigit(c)) {d=d*10+(c - '0'); c=fgetc (fp);}
   else {makelex (3, putnum()); TC = H;}
   break;
        /* .....    */
 } /* конец switch */
 } /* конец тела цикла */
 while (TC != FIN && TC != ER);
  if (TC == ER) printf("ERROR !!!");
  else printf("O.K.!!!");
}
```

Задачи.

33. Дана регулярная грамматика с правилами:

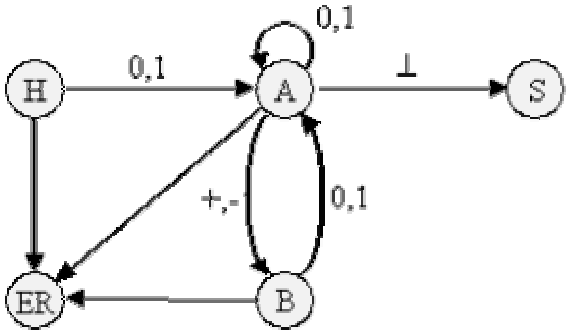
S → S0 | S1 | P0 | P1

P → N.

N → 0 | 1 | N0 | N1 .

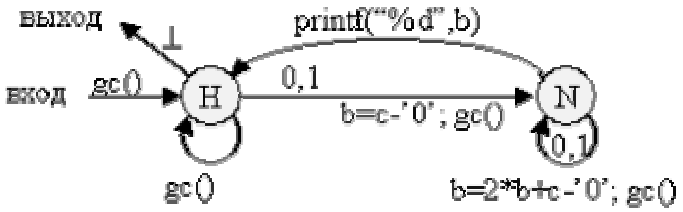
Построить по ней диаграмму состояний и использовать ДС для разбора цепочек : 11.010 , 0.1 , 01. , 100 . Какой язык порождает эта грамматика ?

34. Дана ДС.



1. Осуществить разбор цепочек 1011 , 10+011 и 0-101+1 .
2. Восстановить регулярную грамматику, по которой была построена данная ДС.
3. Какой язык порождает полученная грамматика ?

35. Пусть имеется переменная **c** и функция gc(), считывающая в с очередной символ анализируемой цепочки. Дана ДС с действиями:



1. Определить, что будет выдано на печать при разборе цепочки 1+101//p11+++1000/5□?
2. Написать на Си анализатор по этой ДС.

36. Построить регулярную грамматику, порождающую язык

$L = \{(abb)^k \perp \mid k \geq 1\}$,

по ней построить ДС, а затем по ДС написать на Си анализатор для этого языка.

37. Построить ДС, по которой в заданном тексте, оканчивающемся на ⊥, выявляются все парные комбинации <>, <= и >= и подсчитывается их общее количество.

38. Дана регулярная грамматика:

$S \to A\perp$

$A \to Ab \mid Bb \mid b$

$B \to Aa$

Определить язык, который она порождает; построить ДС; написать на Си анализатор.

39. Написать на Си анализатор, выделяющий из текста вещественные числа без знака (они определены как в Паскале) и преобразующий их из символьного представления в числовое.

40. Даны две грамматики G1 и G2.

G1: $S \to 0C \mid 1B \mid \varepsilon$ G2: $S \to 0D \mid 1B$

$B \to 0B \mid 1C \mid \varepsilon$ $B \to 0C \mid 1C$

$C \to 0C \mid 1C$ $C \to 0D \mid 1D \mid \varepsilon$

$D \to 0D \mid 1D$

$L1 = L(G1)$;

$L2 = L(G2)$.

Построить регулярную грамматику для:

1. L1□L2
2. L1□L2

Если разбор по ней оказался недетерминированным, найти эквивалентную ей грамматику, допускающую детерминированный разбор.

41. Написать леволинейную регулярную грамматику, эквивалентную данной праволинейной, допускающую детерминированный разбор.

a) $S \to 0S \mid 0B$ b) $S \to aA \mid aB \mid bA$

B → 1B | 1C A → bS

C → 1C | ⊥ B → aS | bB | ⊥

c) S → aB d) S → 0B

B → aC | aD | dB B → 1C | 1S

C → aB C → ⊥

D → ⊥

42. Для данной грамматики

1. определить ее тип;
2. какой язык она порождает;
3. написать Р-грамматику, почти эквивалентную данной;
4. построить ДС и анализатор на Си.

S □ 0S | S0 | D

D → DD | 1A | ε

A → 0B | ε

B → 0A | 0

43. Написать анализатор по следующей грамматике:

a) S → C⊥ b) S → C⊥

B → B1 | 0 | D0 C → B1

C → B1 | C1 B → 0 | D0

D → D0 | 0 D → B1

c) S → A0

A → A0 | S1 | 0

44. Грамматика G определяет язык L=L1∪L2, причем L1 ∩L2 =□. Написать регулярную грамматику G1, которая порождает язык L1*L2 (см. задачу 20). Для нее построить ДС и анализатор.

S → 0A | 1S

A → 0A | 1B

B → 0B | 1B | ⊥

45. Даны две грамматики G1 и G2, порождающие языки L1 и L2. Построить регулярные грамматики для

1. L1 □ L2
2. L1 □ L2
3. L1 * L2 (см. задачу 20)

G1: S → 0B | 1S G2: S → B⊥

B → 0C | 1B | ε A → B1 | 0

C → 0B | 1S B → A1 | C1 | B0 | 1

C → A0 | B1

Для грамматики b) построить ДС и анализатор.

46 По данной грамматике G1 построить регулярную грамматику G2 для языка L1* L1 (см. задачу 20), где L1 = L(G1); по грамматике G2 - ДС и анализатор.

G1: S → 0S | 0B

B → 1B | 1C

C → 1C | ε

47. Написать регулярную грамматику, порождающую язык:

1. L = {ω□ | ω □ {0,1}* , где за каждой 1 непосредственно следует 0};
2. L = {1ω1□ | ω □ {0,1}$^+$, где между вхождениями 1 нечетное количество 0};

по ней построить ДС, а по ДС написать на Си анализатор.

48. Построить лексический блок (преобразователь) для кода Морзе. Входом служит последовательность "точек", "тире" и "пауз" (например, ..--. .- ...- ⊥). Выходом являются соответствующие буквы, цифры и знаки пунктуации. Особое внимание обратить на организацию таблицы.

Синтаксический и семантический анализ

На этапе синтаксического анализа нужно установить, имеет ли цепочка лексем структуру, заданную синтаксисом языка, и зафиксировать эту структуру. Следовательно, снова надо решать задачу разбора: дана цепочка лексем, и надо определить, выводима ли она в грамматике, определяющей синтаксис языка. 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 укорачивающие контекстно-свободные грамматики (УКС-грамматики), правила которых имеют вид A □ α, где A □ VN, α □ (VT □ VN)*. Грамматики этого класса, с одной стороны, позволяют достаточно полно описать синтаксическую структуру реальных языков программирования; с другой стороны, для разных подклассов УКС-грамматик существуют достаточно эффективные алгоритмы разбора.

С теоретической точки зрения 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) 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.
Алгоритмы анализа, расходующие на обработку входной цепочки линейное время, применимы только к некоторым подклассам КС-грамматик. Рассмотрим один из таких методов.

Метод рекурсивного спуска

Пример: пусть дана грамматика G =({a,b,c, □}, {S,A,B}, P, S), где

P: $S \to AB\perp$

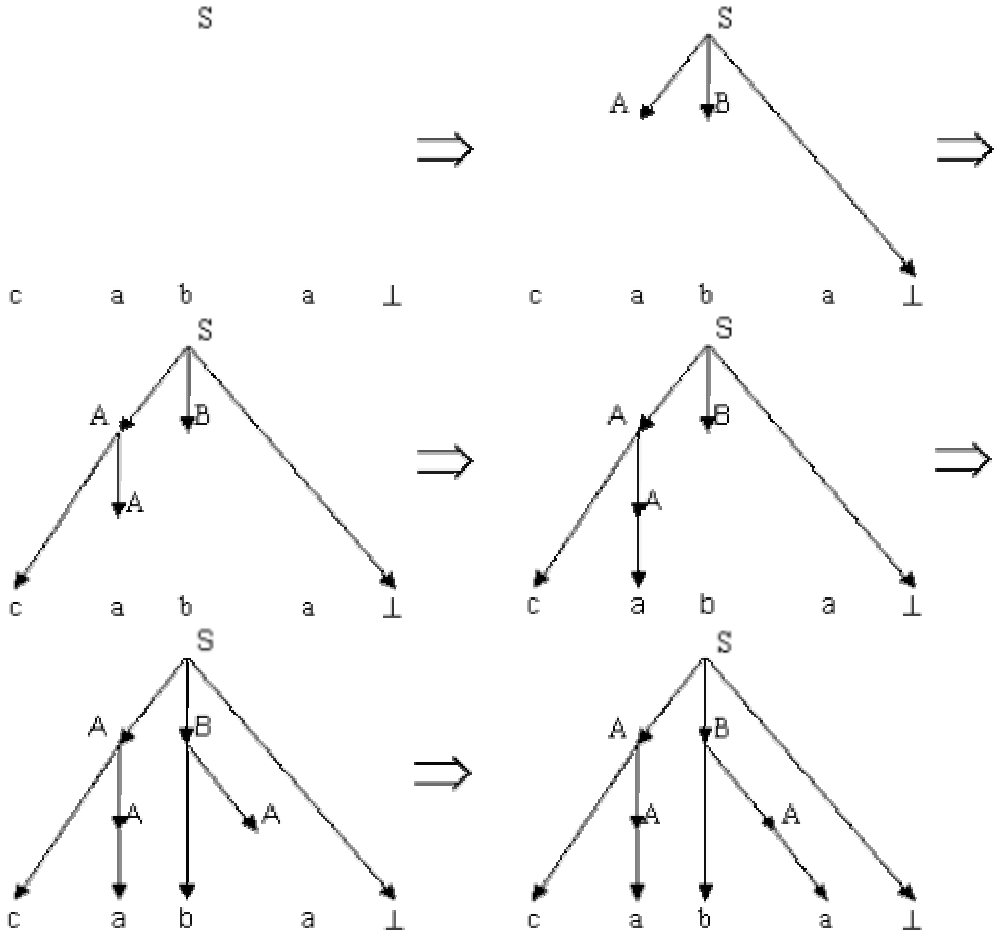
$A \to a \mid cA$

$B \to bA$

и надо определить, принадлежит ли цепочка caba языку L(G).
Построим вывод этой цепочки:

$S \to AB\perp \to cAB\perp \to caB\perp \to cabA\perp \to caba\perp$

Следовательно, цепочка принадлежит языку L(G). Последовательность применений правил вывода эквивалентна построению дерева разбора методом "сверху вниз":



Метод рекурсивного спуска (РС-метод) реализует этот способ практически "в лоб": 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 считать не удается, то процедура завершает свою работу вызовом процедуры обработки ошибки, которая выдает сообщение о том, что цепочка не принадлежит языку, и останавливает разбор. 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.

Пример: совокупность процедур рекурсивного спуска для грамматики G = ({a,b,c,⊥}, {S,A,B}, P, S), где

P: $S \to AB\perp$

$A \to a \mid cA$

$B \to bA$

будет такой:

```
#include <stdio.h>
int c;
FILE *fp; /* указатель на файл, в котором находится
анализируемая цепочка */
void A();
void B();
void ERROR(); /* функция обработки ошибок */
void S() {A(); B();
    if (c != '⊥') ERROR();
      }
void A() {if (c=='a') c = fgetc(fp);
       else if (c == 'c') {c = fgetc(fp); A();}
          else ERROR();
    }
void B() {if (c == 'b') {c = fgetc(fp); A();}
        else ERROR();
       }
void ERROR() {printf("ERROR !!!"); exit(1);}
main() {fp = fopen("data","r");
 c = fgetc(fp);
   S();
     printf("SUCCESS !!!"); exit(0);
    }
```

О применимости метода рекурсивного спуска

Метод рекурсивного спуска применим в том случае, если каждое правило грамматики имеет вид:

1. либо A □ α, где α □ (VT □ VN)* и это единственное правило вывода для этого нетерминала;

2. либо A □ $a_1\alpha_1$ | $a_2\alpha_2$ | ... | $a_n\alpha_n$, где a_i □ VT для всех i = 1,2,...,n; $a_i \neq a_j$ для i ≠ j; α_i □ (VT □ VN)*, т. е. если для нетерминала A правил вывода несколько, то они должны начинаться с терминалов, причем все эти терминалы должны быть различными.

Ясно, что если правила вывода имеют такой вид, то рекурсивный спуск может быть реализован по выше изложенной схеме.
Естественно, возникает вопрос: если грамматика не удовлетворяет этим условиям, то существует ли эквивалентная КС-грамматика, для которой метод рекурсивного спуска применим? К сожалению, нет алгоритма, отвечающего на поставленный вопрос, т.е. это **алгоритмически неразрешимая проблема.**
Изложенные выше ограничения являются достаточными, но не необходимыми. Попытаемся ослабить требования на вид правил грамматики:
(1) 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.).
Общий вид этих правил:

L → a | a,L (либо в сокращенной форме L → a {,a})

Формально здесь не выполняются условия применимости метода рекурсивного спуска, т.к. две альтернативы начинаются одинаковыми терминальными символами.
Действительно, в цепочке a,a,a,a,a из нетерминала L может выводиться и подцепочка a , и подцепочка a,a , и вся цепочка a,a,a,a,a. Неясно, какую из них выбрать в качестве подцепочки, выводимой из L. Если принять решение, что в таких случаях будем выбирать самую длинную подцепочку (что и требуется при разборе реальных языков), то разбор становится детерминированным.
Тогда для метода рекурсивного спуска процедура L будет такой:

```
void L()
{ if (c != 'a') ERROR();
  while ((c = fgetc(fp)) == ',')
    if ((c = fgetc(fp)) != 'a') ERROR();
}
```

Важно, чтобы подцепочки, следующие за цепочкой символов, выводимых из L, не начинались с разделителя (в нашем примере - с запятой), иначе процедура L попытается считать часть исходной цепочки, которая не выводится из L. Например, она может порождаться нетерминалом B - ”соседом” L в сентенциальной форме, как в грамматике

S → LB⊥

L → a {, a}

B → ,b

Если для этой грамматики написать анализатор, действующий РС-методом, то цепочка a,a,a,b будет признана им ошибочной, хотя в действительности это не так.
Нужно отметить, что в языках программирования ограничителем подобных серий всегда является символ, отличный от разделителя, поэтому подобных проблем не возникает.
(2) если грамматика не удовлетворяет требованиям применимости метода рекурсивного спуска, то можно попытаться преобразовать ее, т.е. получить эквивалентную грамматику, пригодную для анализа этим методом.
a) если в грамматике есть нетерминалы, правила вывода которых леворекурсивны, т.е. имеют вид

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

где $\alpha_i \subset (VT \cup VN)^+$, $\beta_j \subset (VT \cup VN)^*$, i = 1,2,...,n; j =1,2,...,m, то непосредственно применять РС-метод нельзя.
Левую рекурсию всегда можно заменить правой:

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

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

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

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

где $a \subset VT$; $\alpha_i, \beta_j \subset (VT \cup VN)^*$, то непосредственно применять РС-метод нельзя. Можно преобразовать правила вывода данного нетерминала, объединив правила с общими началами в одно правило:

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

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

Будет получена грамматика, эквивалентная данной.
c) если в грамматике есть нетерминал, у которого несколько правил вывода, и среди них есть правила, начинающиеся нетерминальными символами, т.е. имеют вид

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

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

. . .

$B_n \to \gamma_{n1} \mid \ldots \mid \gamma_{np},$

где $B_i \subset VN$; $a_j \subset VT$; α_i, β_j, $\gamma_{ij} \subset (VT \cup VN)^*$, то можно заменить вхождения нетерминалов B_i их правилами вывода в надежде, что правило нетерминала A станет удовлетворять требованиям метода рекурсивного спуска:

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

d) если допустить в правилах вывода грамматики пустую альтернативу, т.е. правила вида

$A \to a_1\alpha_1 \mid \ldots \mid a_n\alpha_n \mid \varepsilon$,

то метод рекурсивного спуска может оказаться неприменимым (несмотря на то, что в остальном достаточные условия применимости выполняются).

Например, для грамматики G = ({a,b}, {S,A}, P, S), где

P: $S \to bAa$

$A \to aA \mid \varepsilon$

РС-анализатор, реализованный по обычной схеме, будет таким:

```
void S(void)
{if (c == 'b') {c = fgetc(fp); A();
          if (c != 'a') ERROR();}
 else ERROR();
}

void A(void)
{
   if (c == 'a') {c = fgetc(fp); A();}
}
```

Тогда при анализе цепочки baaa функция A() будет вызвана три раза; она прочитает подцепочку aaa, хотя третий символ a - это часть подцепочки, выводимой из S. В результате окажется, что baaa не принадлежит языку, порождаемому грамматикой, хотя в действительности это не так.

Проблема заключается в том, что подцепочка, следующая за цепочкой, выводимой из A, начинается таким же символом, как и цепочка, выводимая из A.

Однако в грамматике G = ({a,b,c}, {S,A}, P, S), где

P: $S \to bAc$

$A \to aA \mid \varepsilon$

нет проблем с применением метода рекурсивного спуска.

Выпишем условие, при котором ε-правило вывода делает неприменимым РС-метод.

Определение: *множество FIRST(A)* - это множество терминальных символов, которыми начинаются цепочки, выводимые из A в грамматике G = (VT, VN, P, S), т.е. $FIRST(A) = \{ a \subset VT \mid A \Rightarrow a\alpha, A \subset VN, \alpha \subset (VT \cup VN)^* \}$.

Определение: *множество FOLLOW(A)* -это множество терминальных символов, которые следуют за цепочками, выводимыми из A в грамматике

G = (VT, VN, P, S), т.е. FOLLOW(A) = { a ⊂ VT | S ⇒ αAβ, β ⇒ aγ, A ⊂ VN, α, β, γ ⊂ (VT ∪ VN)*}.

Тогда, если **FIRST(A) ∩ FOLLOW(A) ≠ ☐ ,** то метод рекурсивного спуска неприменим к данной грамматике.

Если

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

$B \to \alpha A \beta$

и FIRST(A) ∩ FOLLOW(A) ≠ ☐ (из-за вхождения A в правила вывода для B), то можно попытаться преобразовать такую грамматику:

$B \to \alpha A'$

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

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

Полученная грамматика будет эквивалентна исходной, т.к. из B по-прежнему выводятся цепочки вида α $\{\alpha_i\}$ β_j β либо α $\{\alpha_i\}$ β.

Однако правило вывода для нетерминального символа A’ будет иметь альтернативы, начинающиеся одинаковыми терминальными символами, следовательно, потребуются дальнейшие преобразования, и успех не гарантирован.

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, LR(k)-грамматики, грамматики предшествования и некоторые другие (см., например, [2], [3]).

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.

Соглашение

1. об используемых переменных и типах:

- пусть лексический анализатор выдает лексемы типа struct lex {int class; int value;};
- при описанном выше характере взаимодействия лексического и синтаксического анализаторов естественно считать, что лексический анализатор - это функция getlex с прототипом struct lex getlex (void);
- в переменной struct lex curr_lex будем хранить текущую лексему, выданную лексическим анализатором.

1. об используемых функциях:

int id (void); - результат равен 1, если curr_lex.class = 4, т.е. curr_lex представляет идентификатор, и 0 - в противном случае;
int num (void); - результат равен 1, если curr_lex.class = 3, т.е. curr_lex представляет число-константу, и 0 - в противном случае;
int eq (char * s); - результат равен 1, если curr_lex представляет строку s, и 0 - иначе ;
void error(void) - функция обработки ошибки; при обнаружении ошибки работа анализатора прекращается.
Тогда метод рекурсивного спуска реализуется с помощью следующих процедур, создаваемых для каждого нетерминала грамматики:

для P → program D' ; B⊥

```
void P (void){
  if (eq ("program")) curr_lex = getlex();
  else ERROR();
  D1();
  if (eq (";")) curr_lex = getlex(); else ERROR();
  B();
  if (!eq ("⊥")) ERROR();
}
```

для D' → var D {; D}

```
void D1 (void){
  if (eq ("var")) curr_lex = getlex();
  else ERROR();
  D();
  while (eq (";"))
   {curr_lex = getlex (); D();}
}
```

для D → I {,I}: [int | bool]

```
void D (void){
  if (!id()) ERROR();
  else {curr_lex = getlex();
        while (eq (","))
    {curr_lex = getlex();
          if (!id()) ERROR();
 else curr_lex = getlex ();
          }
  if (!eq (":")) ERROR();
  else {curr_lex = getlex();
        if (eq ("int") || eq ("bool"))
```

```
        curr_lex = getlex();
      else ERROR();}
    }
}
```

для E1 → T {[+ | - | or] T}

```
void E1 (void){
  T();
  while (eq ("+") || eq ("-") || eq ("or"))
   {curr_lex = getlex(); T();}
}
```

. . . . . .

Для остальных нетерминалов грамматики модельного языка процедуры рекурсивного спуска пишутся аналогично.
"Запуск" синтаксического анализатора:

```
... curr_lex = getlex(); P(); ...
```

О семантическом анализе

Контекстно-свободные грамматики, с помощью которых описывают синтаксис языков программирования, не позволяют задавать контекстные условия, имеющиеся в любом языке.
Примеры наиболее часто встречающихся контекстных условий:

1. каждый используемый в программе идентификатор должен быть описан, но не более одного раза в одной зоне описания;
2. при вызове функции число фактических параметров и их типы должны соответствовать числу и типам формальных параметров;
3. обычно в языке накладываются ограничения на типы операндов любой операции, определенной в этом языке; на типы левой и правой частей в операторе присваивания; на тип параметра цикла; на тип условия в операторах цикла и условном операторе и т.п.

Проверку контекстных условий часто называют семантическим анализом. Его можно выполнять сразу после синтаксического анализа, некоторые требования можно контролировать во время генерации кода (например, ограничения на типы операндов в выражении), а можно совместить с синтаксическим анализом.
Мы выберем последний вариант: как только синтаксический анализатор распознает конструкцию, на компоненты которой наложены некоторые ограничения, проверяется их выполнение. Это означает, что на этапе синтаксического анализа придется выполнять некоторые дополнительные действия, осуществляющие семантический контроль.
Если для синтаксического анализа используется метод рекурсивного спуска, то в тела процедур РС-метода необходимо вставить вызовы дополнительных "семантических" процедур (семантические действия). Причем, как показывает практика, удобнее вставить их сначала в синтаксические правила, а потом по этим расширенным правилам

строить процедуры РС-метода. Чтобы отличать вызовы семантических процедур от других символов грамматики, будем заключать их в угловые скобки.

Замечание: фактически, мы расширили понятие контекстно-свободной грамматики, добавив в ее правила вывода символы-действия.

Например, пусть в грамматике есть правило

$A \to a\langle D_1\rangle B\langle D_1; D_2\rangle \mid bC\langle D_3\rangle$,

здесь $A,D,C \subset VN$; $a,b \subset VT$; $\langle D_i\rangle$ означает вызов семантической процедуры D_i, $i = 1, 2, 3$. Имея такое правило грамматики, легко написать процедуру для метода рекурсивного спуска, которая будет выполнять синтаксический анализ и некоторые дополнительные действия:

```
void A() {
  if (c=='a') {c = fgetc(fp); D1(); B(); D1(); D2();}
  else if (c == 'b') {c = fgetc(fp); C(); D3();}
     else ERROR();
}
```

Пример: написать грамматику, которая позволит распознавать цепочки языка $L = \{\alpha \subset (0,1)^+\perp \mid \alpha$ содержит равное количество 0 и 1$\}$.

Этого можно добиться, пытаясь чисто синтаксическими средствами описать цепочки, обладающие этим свойством. Но гораздо проще с помощью синтаксических правил описать произвольные цепочки из 0 и 1, а потом вставить действия для отбора цепочек с равным количеством 0 и 1:

$S \to \langle k0 = 0; k1 = 0;\rangle A\perp$

$A \to 0\langle k0 = k0+1\rangle A \mid 1\langle k1 = k1+1\rangle A \mid$

$0\langle k0 = k0+1;$ check()$\rangle \mid 1\langle k1 = k1+1;$ check()$\rangle$, где

```
void check()
 {if (k0 != k1) { printf("ERROR !!!"); exit(1);}
  else { printf("SUCCESS !!!");exit(0);}
 }
```

Теперь по этой грамматике легко построить анализатор, распознающий цепочки с нужными свойствами.

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.

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. Информацию о типе переменных и о наличии их описания естественно заносить в ту же таблицу.
Пусть каждая строка в TID имеет вид

```
struct record {
  char *name; /* идентификатор */
 int declare; /* описан ? 1-"да", 0-"нет" */
 char *type; /* тип переменной */
 ...
};
```

Тогда таблица идентификаторов TID - это массив структур

```
#define MAXSIZE_TID 1000
struct record TID [MAXSIZE_TID];
```

причем i-ая строка соответствует идентификатору-лексеме вида (4,i).
Лексический анализатор заполнил поле name; значения полей declare и type будем заполнять на этапе семантического анализа.
Для этого нам потребуется следующая функция:
void decid (int i, char *t) - в i-той строке таблицы TID контролирует и заполняет поле declare и, если лексема (4,i) впервые встретилась в разделе описаний, заполняет поле type:

```
void decid (int i, char *t)
 {if (TID [i].declare) ERROR(); /*повторное описание */
 else {TID [i].declare = 1;  /* описан ! */
 strcpy (TID [i].type, t);} /* тип t ! */
 }
```

Раздел описаний имеет вид

D → 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 (например, в стеке), а когда будет

проанализировано имя типа, заполнить поля declare и type в этих строках.
Для этого будем использовать функции работы со стеком целых чисел:

```
void ipush (int i); /* значение i - в стек */
int ipop (void);   /* из стека - целое */
```

Будем считать, что (-1) - "дно" стека; тогда функция

```
void dec (char *t)
 {int i;
   while ((i = ipop()) != -1)
     decid(i,t);
 }
```

считывает из стека номера строк TID и заносит в них информацию о наличии описания и о типе t.
С учетом этих функций правило вывода с действиями для обработки описаний будет таким:

D → **<**ipush (-1)**>** I **<**ipush (curr_lex.value)**>**
{,I **<**ipush (curr_lex.value)**>**}:
[int **<**dec ("int")**>** | bool **<** dec ("bool")**>**]

Контроль контекстных условий в выражении

Пусть есть функция

```
char *gettype (char *op, char *t1, char *t2),
```

которая проверяет допустимость сочетания операндов типа t1 (первый операнд) и типа t2 (второй операнд) в операции op; если типы совместимы, то выдает тип результата этой операции; иначе - строку "no".
Типы операндов и обозначение операции будем хранить в стеке; для этого нам нужны функции для работы со стеком строк:

```
void spush (char *s); /* значение s - в стек */
char *spop (void); /* из стека - строку */
```

Если в выражении встречается лексема-целое_число или логические константы true или false,то соответствующий тип сразу заносим в стек с помощью spush("int") или spush("bool").
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 checkid:

```
void checkid (void)
  {int i;
   i = curr_lex.value;
   if (TID [i].declare)  /* описан? */
spush (TID [i].type); /* тип - в стек */
else ERROR();   /* описание отсутствует */
}
```

Тогда для контроля контекстных условий каждой тройки - "операнд-операция-операнд" будем использовать функцию checkop:

```
void checkop (void)
  {char *op;
char *t1;char *t2;
char *res;
t2 = spop(); /* из стека - тип второго операнда */
op = spop(); /* из стека - обозначение операции */
t1 = spop(); /* из стека - тип первого операнда */
res = gettype (op,t1,t2); /* допустимо ? */
if (strcmp (res, "no")) spush (res); /* да! */
else ERROR();   /* нет! */
}
```

Для контроля за типом операнда одноместной операции *not* будем использовать функцию checknot:

```
void checknot (void)
  {if (strcmp (spop (), "bool")) ERROR();
   else spush ("bool");}
```

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 → E1 | E1 [= | < | >] E1

E1 → T {[+ | - | or] T}

T → F {[* | / | and] F}

F → I | N | [true | false] | not F | (E)

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

Замечание: сравните грамматики, описывающие выражения, состоящие из символов +, *, (,), i:

G1: E → E+E | E*E | (E) | i G4: E → T | E+T

G2: E → E+T | E*T | T T → F | T*F

T → i | (E) F → i | (E)

G3: E → T+E | T*E | T G5: E → T | T+E

T → i |(E) T → F | F*T

F → i | (E),

оцените, насколько они удобны для трансляции выражений.

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

E → E1 | E1 [= | < | >] **<**spush(TD[curr_lex.value])**>**E1 **<**checkop()**>**

E1 → T {[+ | - | or] **<**spush(TD[curr_lex.value])**>**T **<**checkop()**>**}

T → F {[* | / | and] **<**spush(TD[curr_lex.value])**>**F**<**checkop()**>**}

F → I **<**checkid()**>** | N **<**spush("int")**>** |[true | false] **<**spush ("bool") |
not F **<**checknot()**>** | (E)

Замечание: TD - это таблица ограничителей, к которым относятся и знаки операций; будем считать, что это массив

```
#define MAXSIZE_TD 50
char * TD[MAXSIZE_TD];
```

именно из этой таблицы по номеру лексемы в классе выбираем обозначение операции в виде строки.

Контроль контекстных условий в операторах

S ☐ I := E | if E then S else E | while E do S |
B | read (I) | 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 checkid()), то достаточно будет в нужный момент считать из стека два элемента и сравнить их:

```
void eqtype (void)
{if (strcmp (spop (), spop ())) ERROR();}
```

Следовательно, правило для оператора присваивания:

I **<**checkid()**>** := E **<**eqtype()**>**

1. Условный оператор и оператор цикла

if E then S else S | while E 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 eqbool (void)
 {if (strcmp (spop(), "bool")) ERROR();}
```

Тогда правила для условного оператора и оператора цикла будут такими:

if E **<**eqbool()**>** then S else S | while E **<**eqbool()**>** do S

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):

```
#include <string.h>
#define MAXSIZE_TID 1000
#define MAXSIZE_TD 50
 char * TD[MAXSIZE_TD];
 struct record
  {char *name;
 int declare;
   char *type;
 /* ...  */
 };
 struct record TID [MAXSIZE_TID];
 /* описание функций ERROR(), getlex(), id(), eq(char *),
 типа struct lex и переменной curr_lex - в алгоритме
 рекурсивного спуска для М-языка */
 void ERROR(void);
 struct lex {int class; int value;};
 struct lex curr_lex;
 struct lex getlex (void);
 int id (void);
 int eq (char *s);
 void ipush (int i);
 int ipop (void);

 void decid (int i, char *t)
 {if (TID [i].declare) ERROR();
   else {TID [i].declare = 1; strcpy (TID [i].type, t);}
}
 void dec (char *t)
 {int i;
   while ((i = ipop()) != -1) decid (i,t);}
 void D (void)
  {ipush (-1);
   if (!id()) ERROR();
   else {ipush (curr_lex.value);
   curr_lex = getlex ();
        while (eq (","))
        {curr_lex = getlex ();
         if (!id ()) ERROR ();
  else {ipush (curr_lex.value);
   curr_lex = getlex();}
        }
```

```
      if (!eq (":")) ERROR();
   else {curr_lex = getlex ();
         if (eq ("int")) {curr_lex = getlex ();
          dec ("int");}
    else if (eq ("bool"))
       {curr_lex = getlex();
       dec ("bool");}
    else ERROR();
    }
     }
}
```

Задачи.

49. Написать на Си анализатор, действующий методом рекурсивного спуска, для грамматики:

a) S → E⊥

E → () | (E {, E}) | A

A → a | b

b) S → P := E | if E then S | if E then S else S

P → I | I (e)

E → T {+T}

T → F {*F}

F → P | (E)

I → a | b

c) S → type I = T {; I = T} ⊥

T → int | record I: T {; I: T} end

I → a | b | c

d) S → P = E | while E do S

P → I | I (E {, E})

E → E + T | T

T → T * F | F

F → P | (E)

50. Написать для заданной грамматики процедуры анализа методом рекурсивного спуска, предварительно преобразовав ее.

a) S → E⊥

E → E+T | E-T | T

T → T*P | P

P → (E) | I

I → a | b | c

b) S → E⊥

E → E+T | E-T | T

T → T*F | T/F | F

F → I | I^N | (E)

I → a | b | c | d

N → 2 | 3 | 4

c) F → function I(I) S; I:=E end d) S → P := E | while E do S

S → ; I:=E S | ε P → I | I (E {, E})

E → E*I | E+I | I E → E + T | T

T → T * F | F

F → P | (E)

51. Восстановить КС-грамматику по функциям, реализующим синтаксический анализ методом рекурсивного спуска.

```
#include <stdio.h>
int c; FILE *fp;
void A();
void ERROR();
void S (void)
 {if (c == 'a')
  {c = fgetc(fp); S();
   if (c == 'b') c = fgetc(fp);
   else ERROR();
  else A();
}
void A (void)
 {if (c == 'b') c = fgetc(fp);
else ERROR();
  while (c == 'b')
   c = fgetc(fp);
}
void main()
 {fp = fopen("data", "r");
c = fgetc(fp);
  S();
  printf("O.K.!");
}
```

Какой язык она порождает?

52. Предложить алгоритм, использующий введенные ранее преобразования (см. стр. 37-38), позволяющий в некоторых случаях получить грамматику, к которой применим метод рекурсивного спуска.

53. Какой язык порождает заданная грамматика? Провести анализ цепочки (a,(b,a),(a,(b)),b)⊥.

S → **<**k = 0**>** E ⊥

E → A | (**<**k=k+1; if (k == 3) ERROR();**>** E {,E}) **<**k = k-1**>**

A → a | b

54. Есть грамматика, описывающая цепочки в алфавите {0, 1, 2, ⊥}:

S → A⊥

A → 0A | 1A | 2A | ε

Дополнить эту грамматику действиями, исключающими из языка все цепочки, содержащие подцепочки 002.

55. Дана грамматика, описывающая цепочки в алфавите {a, b, c, ⊥}:

S → A⊥

A → aA | bA | cA | ε

Дополнить эту грамматику действиями, исключающими из языка все цепочки, в которых не выполняется хотя бы одно из условий:

- в цепочку должно входить не менее трех букв c ;
- если встречаются подряд две буквы a, то за ними обязательно должна идти буква b.

56. Есть грамматика, описывающая цепочки в алфавите {0, 1}:

S → 0S | 1S | ε

Дополнить эту грамматику действиями, исключающими из языка любые цепочки, содержащие подцепочку 101.

57. Написать КС-грамматику с действиями для порождения
L = $\{a^m b^n c^k \mid m+k = n$ либо $m-k = n\}$.

58. Написать КС-грамматику с действиями для порождения

L = $\{1^n 0^m 1^p \mid n+p > m, m \geq 0\}$.

59. Дана грамматика с семантическими действиями:

S → **<** A = 0; B = 0 **>** L {L} **<** if (A > 5) ERROR() **>** ⊥

L → a **<** A = A+1 **>** | b **<** B = B+1; if (B > 2) ERROR() **>** |
c **<** if (B == 1) ERROR() **>**

Какой язык описывает эта грамматика ?

60. Дана грамматика:

S → E⊥

E → () | (E {, E}) | A

A → a | b

Вставить в заданную грамматику действия, контролирующие соблюдение следующих условий:

1. уровень вложенности скобок не больше четырех;
2. на каждом уровне вложенности происходит чередование скобочных и бесскобочных элементов.

61. Пусть в языке L есть переменные и константы целого, вещественного и логического типов, а также есть оператор цикла

S → for I = E step E to E do S

Включить в это правило вывода действия, проверяющие выполнение следующих ограничений:

1. тип I и всех вхождений E должен быть одинаковым;
2. переменная логического типа недопустима в качестве параметра цикла.

Для каждой используемой процедуры привести ее текст на Си.
62. Дана грамматика

P → program D; begin S {; S} end

D → var D' {; D'}

D'→ I {, I}: record I: R {; I: R} end | I {, I} : R

R → int | bool

S → I := E | I.I := E

E → T {+T}

T → F {*F}

F → I | (E) | I.I | N | L ,

где I - идентификатор, N - целая константа, L - логическая константа.
Вставить в заданную грамматику действия, контролирующие соблюдение следующих условий:

1. все переменные, используемые в выражениях и операторах присваивания, должны быть описаны и только один раз;
2. тип левой части оператора присваивания должен совпадать с типом его правой части.

Замечания: a) все записи считаются переменными различных типов (даже если они имеют одинаковую структуру);
b) допускается присваивание записей.

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:

1. он позволяет фиксировать синтаксическую структуру исходной программы;
2. текст на нем можно автоматически генерировать во время синтаксического анализа;
3. его конструкции должны относительно просто транслироваться в объектный код либо достаточно эффективно интерпретироваться.

Некоторые общепринятые способы внутреннего представления программ:

1. постфиксная запись

2. префиксная запись
3. многоадресный код с явно именуемыми результатами
4. многоадресный код с неявно именуемыми результатами
5. 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 $A, B \subset V_N$.

Выберем в качестве языка для представления промежуточной программы *постфиксную запись* (ее часто называют *ПОЛИЗ* - польская инверсная запись).
В ПОЛИЗе операнды выписаны слева направо в порядке их использования. Знаки операций стоят таким образом, что знаку операции непосредственно предшествуют ее операнды.
Например, обычной (инфиксной) записи выражения
a*(b+c)-(d-e)/f
соответствует такая постфиксная запись:
abc+*de-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.
Более формально постфиксную запись выражений можно определить таким образом:

- если E является единственным операндом, то ПОЛИЗ выражения E - это этот операнд;
- ПОЛИЗом выражения E_1 θ E_2, где θ - знак бинарной операции,

  E_1 и E_2 операнды для θ, является запись E_1' E_2' θ , где E_1' и E_2' - ПОЛИЗ выражений E_1 и E_2 соответственно;

- ПОЛИЗом выражения θ E, где θ- знак унарной операции, а E - операнд θ, является запись E' θ, где E' - ПОЛИЗ выражения E;
- ПОЛИЗом выражения (E) является ПОЛИЗ выражения E.

Запись выражения в такой форме очень удобна для последующей интерпретации (т.е. вычисления значения этого выражения) с помощью стека: выражение просматривается один раз слева направо, при этом

- если очередной элемент ПОЛИЗа - это операнд, то его значение заносится в стек;
- если очередной элемент ПОЛИЗа - это 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;

- когда выражение, записанное в ПОЛИЗе, прочитано, в стеке останется один элемент - это значение всего выражения.

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:

1. заменить унарную операцию бинарной, т.е. считать, что "-a" означает "0-a";
2. 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
I E :=
где ":=" - это двухместная операция, а I и E - ее операнды; 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.**
Введем вспомогательную операцию - условный переход "по лжи" с семантикой
if (not B) then goto L
Это двухместная операция в операндами B и L. Обозначим ее !F, тогда в ПОЛИЗе она будет записана как
B p F!

где p - номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L.
Семантика условного оператора
if B **then** S_1 **else** S_2
с использованием введенной операции может быть описана так:
if (not B) then goto L_2; S_1; goto L_3; L_2: S_2; L_3: ...
Тогда ПОЛИЗ условного оператора будет таким:
B p_2 !F S_1 p_3 ! S_2 ... ,
где p_i - номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L_i, i = 2,3.
Семантика оператора цикла **while** B **do** S может быть описана так:
L_0: if (not B) then goto L_1; S; goto L_0; L_1: ... .
Тогда ПОЛИЗ оператора цикла while будет таким:
B p_1 !F S p_0 ! ... ,
где p_i - номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L_i, i = 0,1.
Операторы ввода и вывода М-языка являются одноместными операциями. Пусть R - обозначение операции ввода, W - обозначение операции вывода.
Тогда оператор ввода **read** (I) в ПОЛИЗе будет записан как I R;
оператор вывода **write** (E) - как E W.
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:

E → T {+T}

T → F {*F}

F → a | b | (E)

Тогда грамматика с действиями по переводу этого выражения в ПОЛИЗ будет такой:

E → T {+T **<**putchar('+')**>**}

T → F {*F **<**putchar('*')**>**}

F → a **<**putchar('a')**>** | b**<**putchar('b')**>** | (E)

Этот метод можно использовать для перевода цепочек одного языка в цепочки другого языка (что, собственно, мы и делали, занимаясь переводами в ПОЛИЗ цепочек лексем).

Например, с помощью грамматики с действиями выполним перевод цепочек языка

$L_1 = \{0^n1^m \mid n,m>0\}$

в соответствующие цепочки языка

$L_2 = \{a^mb^n \mid n,m>0\}$:

Язык L_1 можно описать грамматикой

S → 0S | 1A

A → 1A |ε

Вставим действия по переводу цепочек вида 0^n1^m в соответствующие цепочки вида a^mb^n :

S → 0S **<**putchar('b')**>** | 1 **<**putchar('a')**>** A

A → 1 **<**putchar('a')**>** A |ε

Теперь при анализе 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 (номер_класса, номер_в_классе). Нам придется расширить набор лексем:

1. будем считать, что новые операции (!, !F, R, W) относятся к классу ограничителей, как и все другие операции модельного языка;
2. для ссылок на номера элементов ПОЛИЗа введем лексемы класса 0, т.е. (0,p) - лексема, обозначающая p-ый элемент в ПОЛИЗе;
3. для того, чтобы различать операнды-значения-переменных и операнды-адреса-переменных (например, в ПОЛИЗе оператора присваивания), операнды-значения будем обозначать лексемами класса 4, а для операндов-адресов введем лексемы класса 5.

Будем считать, что генерируемая программа размещается в массиве P, переменная free - номер первого свободного элемента в этом массиве:

```
#define MAXLEN_P 10000
struct lex
 {int class;
```

```
  int value;}
struct lex P [ MAXLEN_P];
int free = 0;
```

Для записи очередного элемента в массив P будем использовать функцию put_lex:

```
void put_lex (struct lex l)
{P [ free++] = l;}
```

Кроме того, введем модификацию этой функции - функцию put_lex5, которая записывает лексему в ПОЛИЗ, изменяя ее класс с 4-го на 5-й (с сохранением значения поля value):

```
void put_lex5 (struct lex l)
{l.class = 5; P [ free++] = l;}
```

Пусть есть функция

`struct lex make_op(char *op),`

которая по символьному изображению операции op находит в таблице ограничителей соответствующую строку и формирует лексему вида (2,i), где i - номер найденной строки.

Генерация внутреннего представления программы будет проходить во время синтаксического анализа параллельно с контролем контекстных условий, поэтому для генерации можно использовать информацию, "собранную" синтаксическим и семантическим 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:

```
void checkop_p (void)
 {char *op; char *t1; char *t2; char *res;
 t2 = spop(); op = spop(); t1 = spop();
  res = gettype (op,t1,t2);
 if (strcmp (res, "no"))
   {spush (res);
    put_lex (make_op (op));} /* дополнение! - операция
 op заносится в ПОЛИЗ */
 else ERROR();
}
```

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

E → E1 | E1 [=|>|<] **<**spush (TD[curr_lex.value])**>** E1
  <checkop_p()**>**

E1 → T {[+|-|or] **<**spush (TD[curr_lex.value])**>**
  T **<**checkop_p()**>**}

T → F {[*|/|and] **<**spush (TD[curr_lex.value])**>**
  F **<**checkop_p()**>**}

F → I **<**checkid(); put_lex (curr_lex)**>** |
  N **<**spush("int"); put_lex (curr_lex)**>** |
  [true|false] **<**spush ("bool"); put_lex (curr_lex)**>** |
  not F **<**checknot(); put_lex (make_op ("not"))**>** | (E)

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

S → I **<**checkid(); put_lex5 (curr_lex)**>** :=

E **<**eqtype(); put_lex (make_op (":="))**>**

При генерации ПОЛИЗа выражений и оператора присваивания элементы массива P заполнялись последовательно. Семантика условного оператора if E then S1 else S2 такова, что значения операндов для операций безусловного перехода и перехода "по лжи" в момент генерации операций еще неизвестны:

if (!E) goto l2; S1; goto l3; l2: S2; l3:...

Поэтому придется запоминать номера элементов в массиве P, соответствующих этим операндам, а затем, когда станут известны их значения, заполнять пропущенное.

Пусть есть функция

struct lex make_labl (int k),

которая формирует лексему-метку ПОЛИЗа вида (0,k).

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

S → if E **<**eqbool(); pl2 = free++; put_lex (make_op ("!F"))**>**

then S **<**pl3 = free++; put_lex (make_op ("!")); P[pl2] = make_labl (free) **>**

else S **<** P[pl3] = make_lable (free) **>**

Замечание: переменные pl2 и pl3 должны быть локализованы в процедуре S, иначе возникнет ошибка при обработке вложенных условных операторов.

Аналогично можно описать способ генерации ПОЛИЗа других операторов модельного языка.

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 записана в массиве P; пусть она состоит из N элементов-лексем. Каждая лексема - это структура

struct lex {int class; int value;},

возможные значения поля class:

1. - лексемы-метки (номера элементов в ПОЛИЗе)
2. - логические константы true либо false (других лексем - служебных слов в ПОЛИЗе нет)
3. - операции (других лексем-ограничителей в ПОЛИЗе нет)
4. - целые константы
5. - лексемы-идентификаторы (во время интерпретации будет использоваться значение)
6. - лексемы-идентификаторы (во время интерпретации будет использоваться адрес).

Считаем, что к моменту интерпретации распределена память под константы и переменные, адреса занесены в поле address таблиц TID и TNUM, значения констант размещены в памяти.
В программе-интерпретаторе будем использовать некоторые переменные и функции, введенные нами ранее.

```
void interpreter(void) {
 int *ip;
 int i, j, arg;
 for (i = 0; i<=N; i++)
  {curr_lex = P[i];
   switch (curr_lex.class) {
  case 0: ipush (curr_lex.value); break;
        /* метку ПОЛИЗ - в стек */
 case 1: if (eq ("true")) ipush (1);
        else ipush (0); break;
   /* логическое значение - в стек */
 case 2: if (eq ("+")) {ipush (ipop() + ipop()); break};
        /* выполнили операцию сложения, результат - в стек*/
 if (eq ("-"))
       {arg = ipop(); ipush (ipop() - arg); break;}
        /* аналогично для других двухместных арифметических
 и логических операций */
 if (eq ("not")) {ipush (!ipop()); break;};
        if (eq ("!")) {j = ipop(); i = j-1; break;};
        /* интерпретация будет продолжена с j-го элемента
            ПОЛИЗа */
 if (eq ("!F")) {j = ipop(); arg = ipop();
             if (!arg) {i = j-1}; break;};
        /* если значение arg ложно, то интерпретация будет
           продолжена с j -го элемента ПОЛИЗа, иначе порядок
           не изменится */
 if (eq (":=")) {arg = ipop(); ip = (int*)ipop();
 *ip = arg; break;};
     if (eq ("R")) {ip = (*int) ipop();
            scanf("%d", ip); break;};
        /* "R" - обозначение операции ввода */
 if (eq ("W")) {arg = ipop();
            printf ("%d", arg); break;};
        /* "W" - обозначение операции вывода */
 case 3: ip = TNUM [curr_lex.value].address;
 ipush(*ip); break;
        /* значение константы - в стек */
```

```
 case 4: ip = TID [curr_lex.value].address;
 ipush(*ip); break;
      /* значение переменной - в стек */
 case 5: ip = TID [curr_lex.value}.address;
 ipush((int)ip); break;
      /* адрес переменной - в стек */
 } /* конец switch */
} /* конец for */
}
```

Задачи.

63. Представить в ПОЛИЗе следующие выражения:
   a) a+b-c b) a*b+c/a
   c) a/(b+c)*a d) (a+b)/(c+a*b)
   e) a and b or c f) not a or b and a
   g) x+y=x/y h) (x*x+y*y < 1) and (x > 0)

64. Для следующих выражений в ПОЛИЗе дать обычную инфиксную запись:
   a) ab*c b) abc*/ c) ab+c*
   d) ab+bc-/a+ e) a not b and not f) abca and or and
   g) 2x+2x*<

65. Используя стек, вычислить следующие выражения в ПОЛИЗе:
   a) xy*xy/+ при x = 8, y = 2 ;
   b) a2+b/b4*+ при a = 4, b = 3 ;
   c) ab not and a or not при a = b = true ;
   d) xy*0 > y2x- < and при x = y = 1 .

66. Записать в ПОЛИЗе следующие операторы языка Си и, используя стек, выполнить их при указанных начальных значениях переменных:
   a) if (x != y) x = x+1 ; при x = 3 ;
   b) if (x > y) x = y ; else y = x ; при x = 5, y = 7;
   c) while (b > a) {b = b-a;} ; при a = 3, b = 7;
   d) do {x = y; y = 2;} while (y > 9); при y = 2;
   e) S = 0; for (i = 1; i <= k; i = i + 1) {S = S + i*i;} при k = 3 ;
   f) switch (k) {
      case 1: a = not a; break;
      case 2: b = a or not b ;
      case 3: a = b ;
      }
      при k = 2, a = b = false .

67. Используя стек, выполнить следующие действия, записанные в ПОЛИЗе, при x = 9, y = 15 (считаем,что элементы ПОЛИЗа перенумерованы с 1).

z, x, y, *, :=, x, y, <>, 30, !F, x, y, <, 23, !F, y, y, x, -, :=, 6, !, x, x, y, -, :=, 6, !, z, z, x, /, :=

Описать заданные действия на Си, не используя оператор goto.

68. Записать в ПОЛИЗе следующие операторы Паскаля:
   a) for I := E1 to E2 do S
   b) case E of
      c1: S1;

```
    c2: S2;
    ....
    cn: Sn
    end;
```
c) repeat S1; S2; ... ;Sn until B;

69. Записать в ПОЛИЗе следующие фрагменты программ на Паскале:

a)
```
case k of
  1: begin a:=not(a or b and c); b:=a and c or b end;
  2: begin a:=a and (b or not c); b:= not a end;
  3: begin a:=b or c or not a; b:==b and c or a end
  end
```
b)
```
S:=0; for i:=1 to N do
begin d:=i*2; a:=a+d*((i-1)*N+5)
S:=-a*d+S
end
```
c)
```
c:=a*b; while a<>b do
if a<b then b:=b-a else a:=a-b;
c:=c/a
```

70. Написать грамматику для выражений, содержащих переменные, знаки операций +, -, *, / и скобки ( ), где операции должны выполняться строго слева направо, но приоритет скобок сохраняется. Определить действия по переводу таких выражений в ПОЛИЗ.

71. Изменить приоритет операций отношения в М - языке (сделать его наивысшим). Построить соответствующую грамматику, отражающую этот приоритет. Написать синтаксический анализатор, обеспечить контроль типов, задать перевод в ПОЛИЗ.

72. Написать КС-грамматику, аналогичную данной,

$E \rightarrow T \{+T\}$

$T \rightarrow F \{*F\}$

$F \rightarrow (E) \mid i$

с той лишь разницей, что в новом языке будет допускаться унарный минус перед идентификатором, имеющий наивысший приоритет (например,
a*-b+-c допускается и означает a*(-b)+(-c).
В созданную грамматику вставить действия по переводу такого выражения в ПОЛИЗ. Для каждой используемой процедуры привести ее текст на Си.

73. Дана грамматика, описывающая выражения:

$E \rightarrow TE'$ $\quad E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$ $\quad T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow PF'$ $\quad F' \rightarrow ^PF' \mid \varepsilon$

$P \rightarrow (E) \mid i$

Включить в эту грамматику действия по переводу этих выражений в ПОЛИЗ. Для каждой используемой процедуры привести ее текст на Си.

74. Написать грамматику для выражений, содержащих переменные, знаки операций +, -, *, /, ** и скобки ( ) с обычным приоритетом

операций и скобок. Включить в эту грамматику действия по переводу этих выражений в префиксную запись (операции предшествуют операндам). Предложить интерпретатор префиксной записи выражений.

75. В грамматику, описывающую выражения, включить действия по переводу выражений из инфиксной формы (операция между операндами) в функциональную запись.

Например,

a+b ==> + (a, b)

a+b*c ==> + (a, * (b, c))

76. Построить регулярную грамматику для языка L1, вставить в нее действия по переводу L1 в L2.

$L1 = \{1^m\ 0^n \mid n,m>0\}$

$L2 = \{1^{m-n} \mid$ если m>n;

0 | если m<n;

ε | если m=n}

(Эта задача аналогична задаче выдачи сообщений об ошибке в балансе скобок).

77. Построить регулярную грамматику для языка L1, вставить в нее действия по переводу цепочек языка L1 в соответствующие цепочки языка L2.

$L1 = \{1^n\ 0^m\ 1^m\ 0^n \mid m,n > 0\}$

$L2 = \{1^m\ 0^{n+m} \mid m,n > 0\}$

78. Построить регулярную грамматику для языка L1, вставить в нее действия по переводу цепочек языка L1 в соответствующие цепочки языка L2.

$L1 = \{b_i \mid b_i = (i)_2$, т.е. b_i -это двоичное представление числа $i \subset N\}$

$L2 = \{(b_{i+1})^R \mid b_{i+1} = (i+1)_2$, ω^R - перевернутая цепочка $\omega\}$

79. Построить грамматику, описывающую целые двоичные числа (допускаются незначащие нули). Вставить в нее действия по переводу этих целых чисел в четверичную систему счисления.