Аргов Д.И.

Основы построения трансляторов

учебное пособие

Рыбинск, 2016 г.

Содержание:

- Транслятор – *это специальная программа, которая переводит исходную программу в эквивалентную ей объектную программу*.
- Если транслятор переводит программу на машинный язык, то он называется компилятор. Интерпретатор же, переводит программу на некоторый промежуточный язык, удобный для специально созданной виртуальной машины. Виртуальная машина – это специальная программа, созданная для выполнения особого набора команд. Каждая команда сначала считывается, затем определяется ее тип, после этого она выполняется.
- Более просто реализовать интерпретатор, так как не придется опускаться до низкоуровневого кодирования на машинном языке. Рассмотрим основные части Интегрированной среды, созданной для реализации нашего языка программирования.

Интегрированная среда содержит:

- Текстовый редактор – в нем пользователь будет набирать исходный текст программы, которая управляет исполнителем.
- Игровое поле – на нем располагаются предметы сбора, препятствия, противники и т.д.
- Меню – система управления средой (запуск программы, выход и т.д.)
- Транслятор – переводит программу из текстового вида (набрал пользователь) в объектный вид (особый список), удобный для исполнения.

Интегрированная среда

Меню

Исполнитель

Предмет сбора

Текстовый редактор

Препятствие

Игровое поле

Меню загрузки

Количество предметов

Общий алгоритм работы транслятора

Исходный текст программы

Сканер – разбивает текст на лексемы

Генератор кода – преобразует лексемы в список

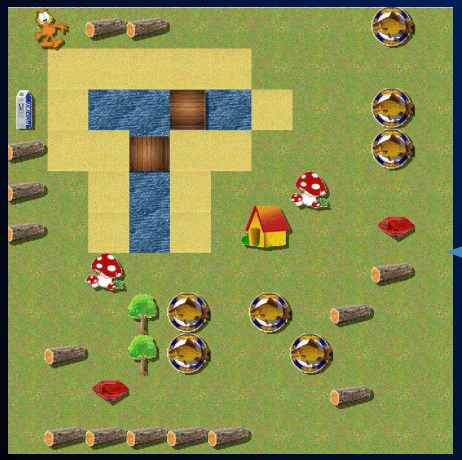
Переменные

ВПРАВО
ВЗЯТЬ
ВНИЗ
…

cmRight
СКАНЕР
cmTake
cmLeft

Генератор кода

cmRight
cmTake
cmLeft



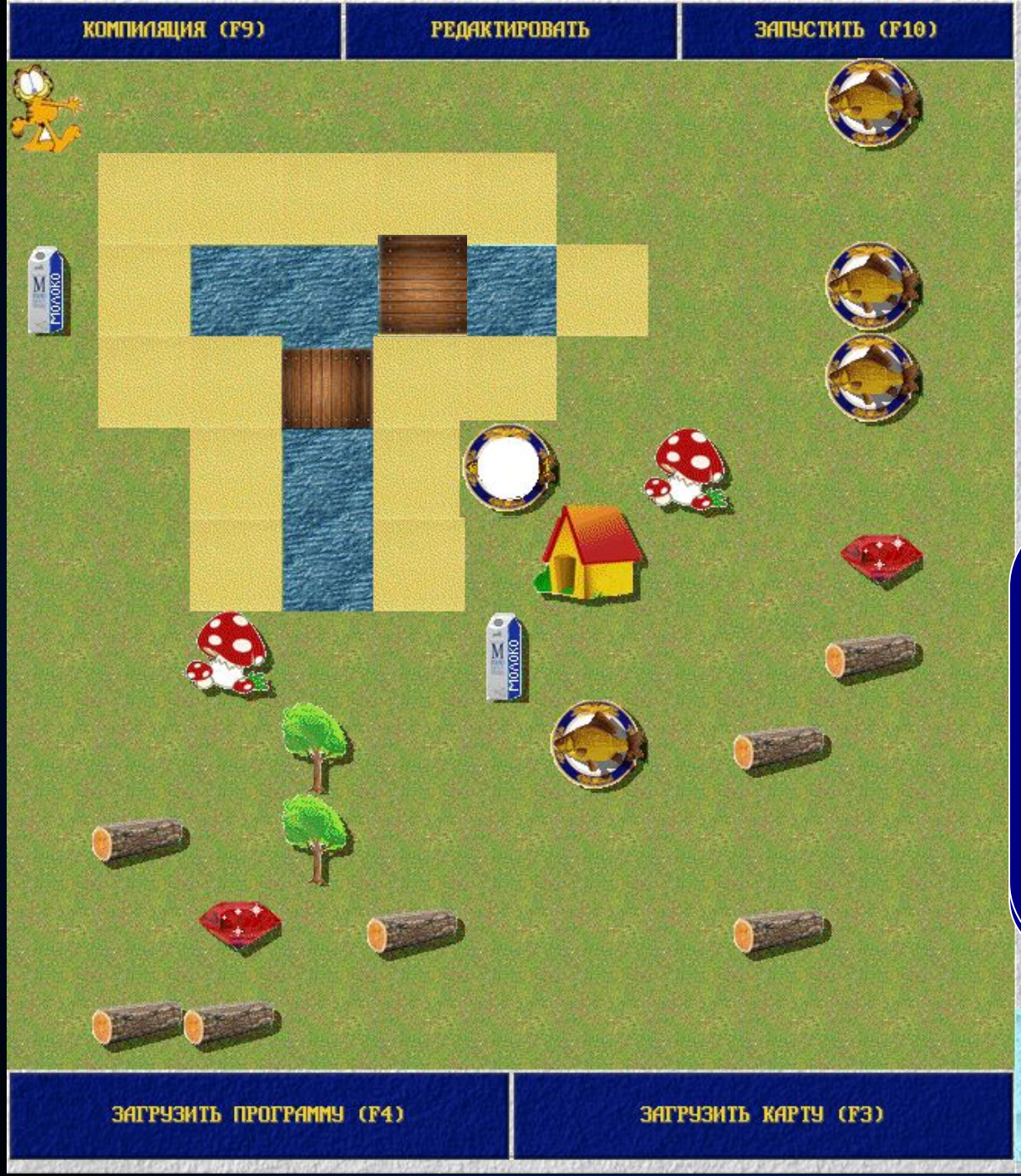
Матрица поля

0	25	3
1	0	2
0	10	0

Интерпретатор

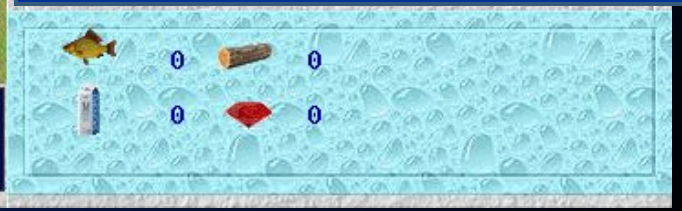
Интерпретатор – исполняет объектный код

- **Сканер** – лексический анализатор, который выделяет лексемы из исходного текста программы. Лексема – это последовательность символов, задающая либо команду, либо разделитель (“;”, “:”), либо операцию (“+”, “-“, “:=”) и т.д. Лексемы поступают в блок синтеза или **генератор кода**. Его задача – получить лексему, понять ее смысл, и создать объектный блок, который будет соответствовать этой лексеме. Простота построения транслятора зависит от выбранного языка. Чем более продуман язык, тем проще транслятор. В процессе работы генератора кода, формируются специальные таблицы:
- Таблица переменных и их типов;
- Таблица констант;
- Возможно формирование таблицы типов.




```
ПРОГРАММА ТЕСТ;
ПЕРЕМЕННЫЕ И:ЦЕЛЫЙ;
НАЧАЛО
  НАЧАТЬ_С 0 0 2;
  ДЛЯ И=1 ДО 4
  НАЧАЛО
     ВЗЯТЬ;
     ВПРАВО;
 КОНЕЦ;
КОН_ДЛЯ;
```

Как можно заметить, некоторым командам исходной программы соответствуют свои действия исполнителя: переместиться, взять предмет, построить мост и т.д.



Сканер

Как уже говорилось ранее, сканер – это лексический анализатор, предназначенный для просмотра исходного текста программы и выделения лексем.

Например, пусть имеется некоторая программа:

ПРОГРАММА ИМЯ;
ПЕРЕМЕННЫЕ
    A, B: **ЦЕЛЫЙ**;
НАЧАЛО
    A:=5;
…

После начала работы сканер будет выделять лексемы в следующем порядке: «ПРОГРАММА». Получив ее, генератор кода проверит лексему по специальной таблице команд. Генератор кода знает, что за этой лексемой должен следовать идентификатор. Если его нет, то, значит, возникла ошибка. Получив лексему «ПЕРЕМЕННЫЕ», генератор кода будет считывать идентификаторы переменных и помещать их в специальный временный буфер. Процесс будет продолжаться до тех пор, пока не появится лексема ”:”. Ее появление означает начало объявления типа. Все имена переменных переписываются из временного буфера в специальную таблицу имен переменных, где хранится: имя переменной, ее тип, текущее значение. Для простоты можно рассмотреть только целые переменные.

```
Type
Str_15=string[15];
tVarName=array[1..50] of Str_15;
{Тип временного буфера имен переменных}
tVar=record {Тип элемента таблицы имен переменных}
            Name:Str_15;   {Имя переменной}
            Znach:integer;{Ее текущее значение}
            Type_:Ident;   {Тип переменной}
          end;
tVarM=array[1..50] of tVar;
{Тип таблицы имен переменных}
Var
 VarN:tVarM;{Таблица имен переменных}
```

- **Рассмотрим подпрограмму GetLex, которая непосредственно выделяет лексемы из входного файла, и помещает лексему в глобальную переменную Ch.**

```
Function GetLex:Ident;
var i,Code:integer;
    ii:Ident;
begin
   While  St='' do
     ReadLn(f,st);{стр
   i:=1;
   while st[i]=' ' do
     inc(i);
   delete(st,1,i-1);
   i:=1;
   case st[1] of {анализируем первый символ}
    '<','>': begin
                 if st[2] in ['>','=']
                 then i:=2
                 else i:=1;
                 Lex:=Copy(st,1,i);
                 Delete(st,1,i);
               end;
```

Сканер определяет, что обнаружен символ ”<”, но пока это ничего не значит, так как еще не известно, что следует за ним. Может далее идет символ “=”, а он существенно меняет смысл знака “<”. Поэтому, программа проверяет второй символ. Вырезанная лексема, помещается в переменную Lex. Далее в специальном цикле, будет просматриваться таблица идентификаторов и в ней выбираться соответствующий лексеме идентификатор.

```
':',';',',','=': {Здесь выделяются однозначковые лексемы}
              begin
                Lex:=Copy(st,1,1);
                Delete(st,1,1);
               end;
   '0'..'9':begin {Здесь выделяется число}
               while st[i] in ['0'..'9','.'] do
                 inc(i);
              Lex:=Copy(st,1,i-1);
              Val(Lex,LexNum,code);
              Delete(st,1,i-1);
              GetLex:=cmNumber;
              exit;
             end;
  else {Если ничего из выше перечисленного нет, то это}
      if st[1] in ID   {идентификатор ID
      then begin       {из которого
              while st[i] in I
                inc(i);
              Lex:=Copy(st,1,i
              Delete(st,1,i-1)
           End
      Else Error('Недопустимый
  end;{case}
```

Теперь в переменной Lex хранится лексема в строковом виде. Однако работать с ней в таком виде очень неудобно. Поэтому, мы введем специальный перечислимый тип Ident. В нем будут перечислены все возможные типы идентификаторов. Порядок следования имен в типе Ident должен полностью соответствовать порядку следования имен в массиве MainLex

```
 ii:=cmProg;  {Начинаем с первого идентификатора}
  while (MainLex[ii]<>Lex)and(byte(ii)<MaxLex) do
     inc(ii);   {пока имена не совпали, двигаемся дальше}
     GetLex:=ii  {возвращаем код идентификатора}
end;
```

Рассмотрим объявление типов

```
Type Ident = (cmProg, cmIf, cmThen, cmElse, cmEndIf,
cmBegin, cmEnd, cmFor, cmTo, cmInt, cmTZ, cmTT,
cmZP,cmLeft, cmRight, cmEqu, cmNumber,cmLess,
cmMore,cmNotEqu,cmLessEQU, cmMoreEqu, cmIdent);
const MaxLex=22;
      MainLex:array[ident] of Str_15=
   ('ПРОГРАММА','ЕСЛИ','ТОГДА','ИНАЧЕ', 'КОН_ЕСЛИ',
   'НАЧАЛО', 'КОНЕЦ', 'ДЛЯ', 'ДО', 'ЦЕЛЫЙ', ';', ':',
   ',', 'ВЛЕВО', 'ВПРАВО','=','','<','>',
   '<>','<=','>=', '');
```

Коду CmProg соответствует слово ‘ПРОГРА
выделенное слово “ПРОГРАММА”, совпа
массива MainLex, то цикл прервется:

```
while (MainLex[ii]<>Lex)and(byte(i
      inc(ii); {пока имена не совпали, дв
```

а переменная ii будет равна коду cmProg.

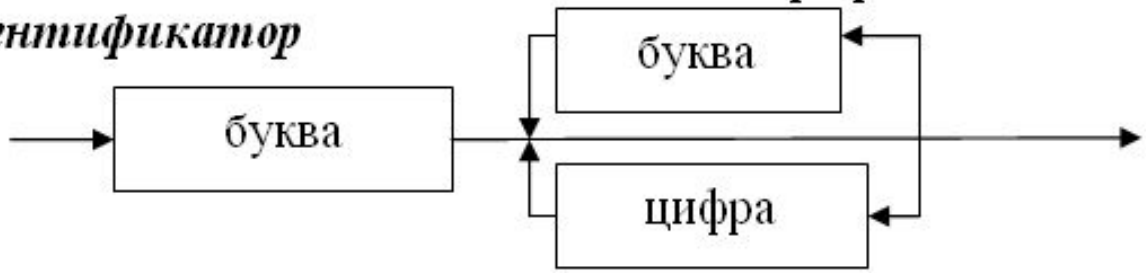
Самым последним значением в перечислимом типе Ident должно быть значение cmIdent – идентификатор. Это значение не имеет эквивалента в строковом массиве MainLex.

Грамматика языка

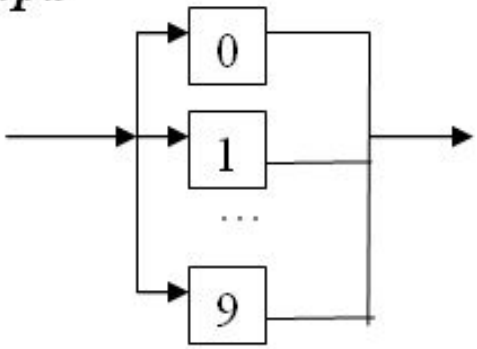
- Прежде, чем создавать генератор кода, необходимо разработать язык исходной программы. От того, как вы продумаете входной язык, зависит очень многое, в частности, простота построения генератора кода, удобство для пользователя. Например, можно выбрать русские команды, но сразу возникают проблемы с русификатором, английский же язык неудобен для маленьких пользователей.
- Входной язык принято описывать специальным образом. Существует несколько способов описания:
- **Описание с помощью БНФ (нормальные формы Бекуса Наура)**
- <идентификатор>::=<буква>|<идентификатор><цифра>|<идентификатор><буква>
- <константа>::=<знак>|<цифра>|<точка>|<константа>
- знак “|” означает ИЛИ, “::=” означает “ПО ОПРЕДЕЛЕНИЮ ЕСТЬ”

- **Описание с использованием графов:**

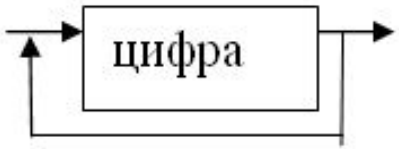
идентификатор



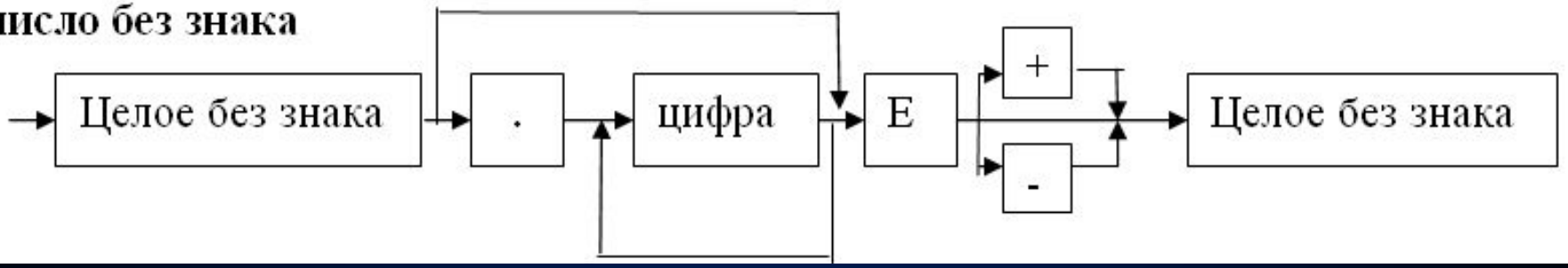
цифра



целое без знака



число без знака



Существует два способа разбора языков: восходящий и нисходящий. Мы только что рассмотрели восходящий. Сначала описываются некоторые простые элементы, затем более сложные, причем при их описании могут использоваться простые. О нисходящих грамматиках поговорим подробнее.

Метод рекурсивного спуска

Пусть имеется грамматика, описывающая некоторый язык:

<программа>::=ПРОГРАММА <имя;>|<раздел переменных><тело программы>

<раздел переменных>::= <имя>|<, имя> <:> <тип переменной;>|<раз

{повтор «раздел пере ти го
что описа

<тип перем

<имя>::=<б

<тело прог

<оператор>
<ВНИЗ>
<операто

Не буду опи
БНФ опер
простой и
аналоги паскалевских For и While.

В описании языка участвовали два типа символов:

те, которые написаны маленькими буквами, называются нетерминальными и непосредственно в программе не используются, однако, за каждый из них будет отвечать своя подпрограмма. Вспомните, мы говорим «в программе ова», есть оператор ветвления», причем можем не уточнять какой именно, полный или сокращенный. Термин «оператор ветвления» - это и есть нетерминальный символ, он состоит из терминальных: «ЕСЛИ», «ТОГДА», «ИНАЧЕ».

```
ПРОГРАММА  МОЕ_ТВОРЕНИЕ;
  У, И, Х: ЦЕЛОЕ;
НАЧАЛО
    ВЛЕВО;
    ВПРАВО;
    ДЛЯ И=1 ДО 10 {цикл с известным чис
      НАЧАЛО
        ВЛЕВО;
        ВПРАВО;
      КОНЕЦ;
    ЕСЛИ Х<5    {оператор ветвления}
    ТОГДА НАЧАЛО   {ветка «Да», составной оператор}
           ВЛЕВО;
           ВПРАВО;
         КОНЕЦ
    ИНАЧЕ   ВЛЕВО;     {ветка «Нет», простой оператор}
    КОН_ЕСЛИ;  {конец оператора ветвления}
КОНЕЦ.  {конец программы}
```

Итак, начнем… Первым ключевым словом в программе должно быть слово «ПРОГРАММА», Затем возможен раздел описания переменных или тело программы. Нашей первой строке грамматике в БНФ будет соответствовать отдельная процедура.

нет.

```
Procedure progr; {считаем, что лексема уже считана в Ch}
begin
 if Ch<>cmProg {Если первая лексема не ПРОГРАММА, то ошибка}
   then Error('Требуется слово ПРОГРАММА')
   else begin
          Ch:=GetLex; {Считать очередную лексему}
          if Ch<>cmIdent {Если не «имя»,то ошибка}
          then Error('Требуется имя программы')
          else begin
                 Ch:=GetLex;
                 if Ch<>cmTZ then Error('Требуется «;»');
                 Ch:=GetLex;
               end
        end
end;
```

В самой программе данная процедура вызывается так:

```
Ch:=GetLex;{считать первую лексему}
Progr; {проанализировать раздел имени программы}
if Ch=cmIdent {если лексема=идентификатор, то это}
Then RazdelVar;{раздел описания переменных}
if ch=cmBegin  {Если лексема=НАЧАЛО, то разобрать}
then  Operator(Last); {тело программы}
```

За каждый раздел (описания переменных и тело программы) отвечает своя процедура, только она знает, как должен выглядеть этот раздел.

```
procedure RazdelVar;
var i:integer;
begin
Repeat {первый цикл нужен, так как возможно несколько
 групп переменных, каждая своего типа. }
   Repeat {второй цикл идет до ":", выделяет имена
  переменных}
     if ch=cmIdent {Если это имя, то запомнить его}
     then begin
            inc (NumbV); VarName[NumbV]:=Lex;
          end;
     Ch:=GetLex;
     if not(Ch in [cmTT,cmZP]){cmTT=':', cmZP=','}
     then Error('Требуется :')
     else if Ch=cmZP then  Ch:=GetLex;
   until Ch=cmTT;{закончить, когда дойдем до ":"}
```

```
 Ch:=GetLex;
 if not (Ch in[cmInt]){если не указан тип, то ошибка}
 then Error('Требуется указать тип переменной')
 else begin
 {переписываем найденные имена в таблицу имен переменных}
       for i:=1 to NumbV do
        begin
          inc(NumbVar);
          VarN[NumbVar].Name:=VarName[i];
          VarN[NumbVar].Type_:=Ch; {cmInt}
          VarN[NumbVar].Znach:=0;
        end;
       NumbV:=0;
    end;
 Ch:=GetLex;
 if (Ch <>cmTZ)then Error('Требуется ;')
 else Ch:=GetLex;
until Ch=cmBegin {Раздел описания переменных закончится словом
   «НАЧАЛО», а это уже не «наша» компетенция}
end;
```

```
Repeat
  Repeat
   if ch=cmIdent
   then begin
          inc (NumbV);VarName[NumbV]:=Lex;
        end;
    Ch:=GetLex;
    if not(Ch in [cmTT,cmZP])
    then Error('Требуется :')
    else if Ch=cmZP then  Ch:=GetLex;
 until Ch=cmTT;{закончить, когда дойдем до ":"}
 Ch:=GetLex;
 if not (Ch in[cmInt])
 then Error(' Требуется указать тип переменной')
 else begin
       for i:=1 to NumbV do
        begin
          inc(NumbVar);
          VarN[NumbVar].Name:=VarName[i];
          VarN[NumbVar].Type_:=Ch; {cmInt}
          VarN[NumbVar].Znach:=0;
        end;
       NumbV:=0;
    end;
 Ch:=GetLex;
 if (Ch <>cmTZ)then Error('Требуется ;')
 else Ch:=GetLex;
until Ch=cmBegin
```

```
A, B, C: ЦЕЛЫЕ;
X: ДРОБНЫЕ;
НАЧАЛО
```

Lex: НАЧАЛО

Ch: cmBegin

VarName

1	2	3	4	5	6
A	B	C	X		

VarN

1	2	3	4	5	6
A	B	C	X		
Ц	Ц	Ц	Д		
0	0	0	0		

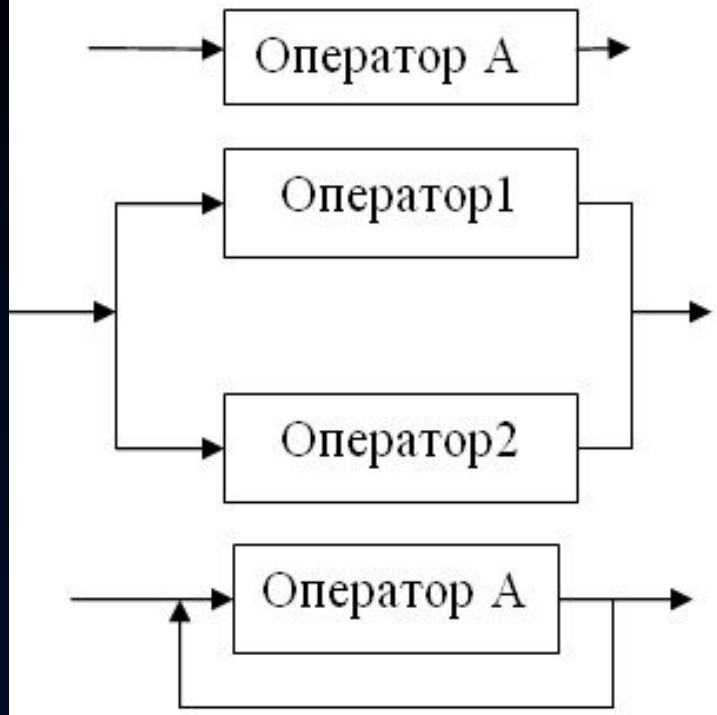
Встретив лексему НАЧАЛО, цикл разбора раздела описания переменных заканчивается. Дальше будет работать другая подпрограмма – разбор операторов.

Сложнее разобрать тело программы.

<тело программы>::=<НАЧАЛО> <оператор> <КОНЕЦ> <.>

<оператор>::=<ВЛЕВО><;>|<ВПАРВО><;>|<ВВЕРХ><;>|
<ВНИЗ><;>|
<оператор ветвления><;>| <оператор
цикла><;>|<оператор>

Оно начинается ключевым словом НАЧАЛО, заканчивается - словом КОНЕЦ. Само тело программы может состоять из нескольких операторов, причем их количество, порядок расположения заранее не известен. Самое «неприятное» состоит в том, что некоторые операторы могут содержать в себе другие вложенные операторы. Например, оператор ветвления может содержать другой оператор ветвления. Как же осуществить перевод грамматики языка из графического представления в программный код, который будет распознавать исходный текст программы.



Вызов процедуры, выполняющей команду А

Выбор операторов:
Case Ch of
Оператор1: вызов процедуры;
Оператор2: вызов процедуры2;
end;

repeat
Оператор А;
until Ch in [завершающая лексема] do

- Для каждого оператора есть своя начальная и завершающая лексема:

Оператор	Начальная лексема	Конечная лексема
Ветвления	ЕСЛИ	КОН_ЕСЛИ
Цикл с известным числом повторений	ДЛЯ	КОНЕЦ или КОН_ДЛЯ
Цикл с предусловием	ПОКА	КОНЕЦ или КОН_ПОКА

Рассмотрим подпрограмму, которая разбирает тело программы.

```
В цикле пока не встретится лексема=КОНЕЦ;
    Считать очередную лексему в переменную Ch;
    Case Ch of
      cmLeft: обработать команду влево;
        cmRight: обработать команду вправо;
        cmIf: обработать оператор ветвления;
    end;
```

За обработку каждого оператора отвечает своя подпрограмма, она создает новое звено, вставляет его в цепочку объектного кода, настраивает его параметры (например, начальное и конечное значение счетчика цикла).

```
Procedure Operator(var Last:pNode);forward;
```
{это описание позволит вам ссылаться на подпрограмму, которая будет описана позже}

…{здесь необходимо описать подпрограммы разбора различных операторов, некоторые из них (например, оператор ветвления) потребуют вызова еще неописанной процедуры Operator. А она, в свою очередь, потребует вызова процедуры разбора оператора ветвления.}

```
Procedure Operator(var Last:pNode); {Переменная Last используется как
   указатель на последнее звено цепочки объектного кода, но об этом позже}
begin
    repeat    {цикл пока не дошли до лексемы КОНЕЦ}
      Ch:=GetLex; {считать очередную лексему}
      case Ch of
        cmIf: OperIf(Last);{Если это оператор ветвления, то разобрать его}
        cmFor: OperFor(Last);{Если это цикл, то разобрать его}
        cmLeft:begin   {Если это команда ВЛЕВО, то создать звено }
                 AddElem(Last,NewElem(cmLeft));{и разместить его в}
                 Ch:=GetLex;{цепочке  объектного кода}
               end;
        cmRight:begin
                  AddElem(Last,NewElem(cmRight));
                  Ch:=GetLex;

      end;{case
      if Ch<>cm                                  ошибка
      then Erro
   until Ch=cmE
end;
```

Из примера видно, что довольно нелогичная структура составного оператора паскаля begin… end становится более логичной. Эту подпрограмму Operator можно использовать не только для разбора основного тела программы, но и для разбора операторов веток «Да» и «Нет» оператора ветвления, тела цикла… Ведь составной оператор (тело цикла) можно рассматривать как новое маленькое тело программы. Поэтому подпрограмма «оператор ветвления» может вызывать подпрограмму «оператор», а она – его, то есть на лицо косвенная рекурсия.

Генератор кода

Генератор кода – это часть транслятора, которая отвечает за создание объектного кода – программы, представленной в удобном для исполнения виде. Наш объектный код может быть представлен в двух формах:

- В виде цепочки объектов (для 3 года обучения);
- В виде цепочки динамических звеньев. Этот способ менее удобен, но, как говорится, «На безрыбье и рак - рыба» (см. Динамические списки).

Каждый оператор исходного языка будет однозначно представлен записью с несколькими полями. Рассмотрим самый простой случай, когда все поля записи хранятся вне зависимости от типа звена (например, команде «ВЛЕВО» указатель на ветку «Да» не нужен).

```
pNode=^tNode;      {pNode – указатель на объект-звено}
tNode=record  {Само звено}
     Typ:Ident;     {  Тип звена, говорит интерпретатору что делать}
     Then_,Else:pNode;{Указатели на цепочки операторов веток «Да» и «Нет»}
     Next: pNode; {Указатель на следующее звено}
     op1,op2:pZnach;  {два операнда из условия оператора ветвления}
     Operation:Ident;{операция, использованная в условии опер. ветв.}
     ForI:integer;     {Указатель на счетчик цикла For}
end;
```

Как видно из примера, многие поля не будут использоваться в каждом звене, как этого избежать, мы узнаем позже. Обратите внимание на то, что в паскале есть особый тип – запись с вариантами, он более эффективен в данном случае.

```
Type pObject=^tObject;
     tPoint=record
      Case Target:boolean of
       True: (tp:pObject);
       False: (Tx,Ty: integer);
     End;
```

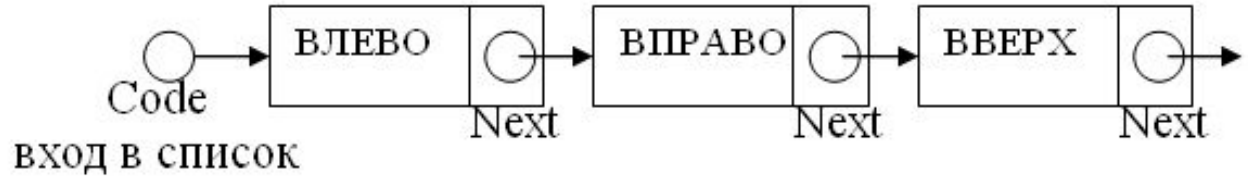
Данная структура называется запись с вариантами, она позволяет использовать несколько групп полей, которые не нужны одновременно (что позволяет уменьшить размер переменной и сэкономить память). У данной записи есть обязательное поле Target *(существует всегда). В зависимости от его значения появляются другие поля:*

Если Target=true, то существует поле tp – указатель на объект, который мы атакуем.

Если Target=false, то существуют два поля tx и ty – координаты цели (например, куда ехать)

Одновременно tp и tx, ty не могут существовать!

ВЛЕВО;
ВПРАВО;
ВВЕРХ;
…



ВНИЗ;
ЕСЛИ X<5
ТОГДА НАЧ
КОН
ИНАЧЕ ВНИ
КОН_ЕСЛИ;
ВВЕРХ;

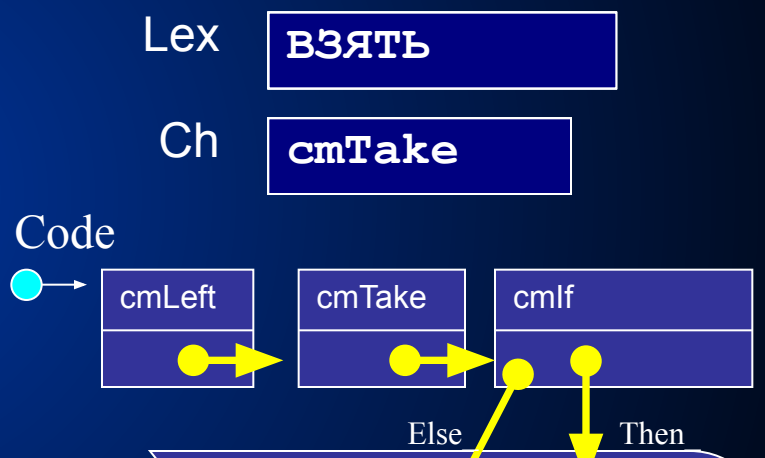
У оператора цикла есть управляющее звено, которое хранит: индекс переменной-счетчика, начальное и конечное значение счетчика, указатель на список операторов, являющихся телом цика. Этот список может содержать любые операторы, в том числе и другой цикл.

ВВЕРХ;
ДЛЯ И=1 Д
НАЧАЛО
ВНИЗ
ВЛЕВ
КОНЕЦ;
ВЛЕВО;

У оператора ветвления есть заглавное звено, которое хранит условие и два указателя: на ветку «да» и ветку «нет». Каждая ветка – это аналогичный ВЛЕВО nil, который может состоять из одного или нескольких звеньев. Если оператор сокращенный, то указатель Else_=nil. Для удобства работы списки чаще всего делают с заглавным элементом (на рисунке не показано)

```
repeat
 Ch:=GetLex;
 case Ch of
  cmIf: OperIf(Last);
  cmFor: OperFor(Last);
   cmLeft:begin
     AddElem(Last,NewElem(cmLeft));
     Ch:=GetLex;
   end;
   cmRight:begin
     AddElem(Last,NewElem(cmRight));
     Ch:=GetLex;
    end;
cmTake:begin
     AddElem(Last,NewElem(cmTake));
     Ch:=GetLex;
   end;
 end;{case}
 if Ch<>cmTZ
 then Error('Требуется «;»'
until Ch=cmEnd;
```

```
ВЛЕВО;
ВЗЯТЬ;
ЕСЛИ X=0
ТОГДА ВПРАВО;
ИНАЧЕ НАЧАЛО
        ВЛЕВО;
        ВЗЯТЬ;
КОН_ЕСЛИ
```



Читаем очередной идентификатор (
за его разбор отвечает своя отдел
процедура OperIf. Она разбирает условие
и сохраняет его в звене, з зби
ветку ТОГДА и прицепляе аза
Then_, если есть ветка ИНАЧЕ, то
разбирает и ее

Оператор ветвления

```
Procedure OperIf(var Last:pNode); {Анализ оператора ветвления}
Var pp:pNode;
begin
   AddElem(Last,NewElem(cmIf)); {создать новое звено}
{Процедура NewElem создает новое звено указанного типа, AddElem – вставляет созданное звено
   в цепочку объектного кода, смещает указатель Last так, чтобы он всегда указывал на
   последнее звено}
   Uslovie(Last);    {выделить условие, разместить его в созданном звене}
   Ch:=GetLex;
   if Ch<>cmThen  {Если после условия нет ветки ТОГДА, то ошибка}
   then Error('Требуется «ТОГДА»');
   Ch:=GetLex;    New(Last^.Then_); {Создать заглавный элемент ветки «Да»}
   pp:=Last^.Then_;
   if Ch=cmBegin          {Если после лексемы «ТОГДА» идет слово НАЧАЛО, то}
   then Operator(pp)    {обработать составной оператор}
   else SimpOper(pp); {иначе простой (без цикла repeat … until Ch=cmEnd)}
   Ch:=GetLex;
   if Ch=cmElse  {анализ возможной ветки ИНАЧЕ}
   then begin
          Ch:=GetLex;  New(Last^.Else_); pp:=Last^.Else_;
          if Ch=cmBegin then Operator(pp) else SimpOper(pp);
          Ch:=GetLex;
        end;
   if Ch<>cmEndIf then Error('Требуется Кон_если');
   Ch:=GetLex;
end;
```

Каждое звено будет хранить одинаковые поля, часть из них вообще не будет использоваться. Чтобы это избежать, можно воспользоваться особым типом языка Паскаль – записью с вариантами. Название и тип полей будет определяться значением одного поля – Typ.

```
pNode=^tNode;   {pNode – указатель на объект-звено}
tNode=record{Само звено}
  Next: pNode; {Указатель на следующее звено}
  Case Typ:Ident of {Тип звена, говорит интерпретатору что делать}
   CmIf:( Then_, Else_: pNode;
                {Указатели на цепочки операторов веток «Да» и «Нет»}
      op1,op2:pZnach;{два операнда из условия оператора ветвления}
      Operation:Ident;{операция, использованная в условии опер. ветв.}
    );
    CmFor:(ForI: integer; {Указатель на счетчик цикла For}
           Body: pNode; {Указатели на тело цикла}
    c
end;
```

Обращаю внимание на то, что некоторые поля не могут использоваться одновременно, их существование определяется значением поля Typ. Например, если Typ=cmIf, то есть поле Then_, но нет поля Body. Хотя пользователь может обратиться к этому полю, компилятор этой ошибки обнаружить не может. Оператор Case в записи может использоваться только в конце, после описания общих полей. Слово End ставится только одно для записи и оператора Case.

Поля op1,op2 имеют тип pZnach. Это тоже указатель на особую запись. Дело в том, что в условии оператора ветвления могут участвовать как числа, так и переменные:

ЕСЛИ X<У …

ЕСЛИ X<0 …

ЕСЛИ 0<У …

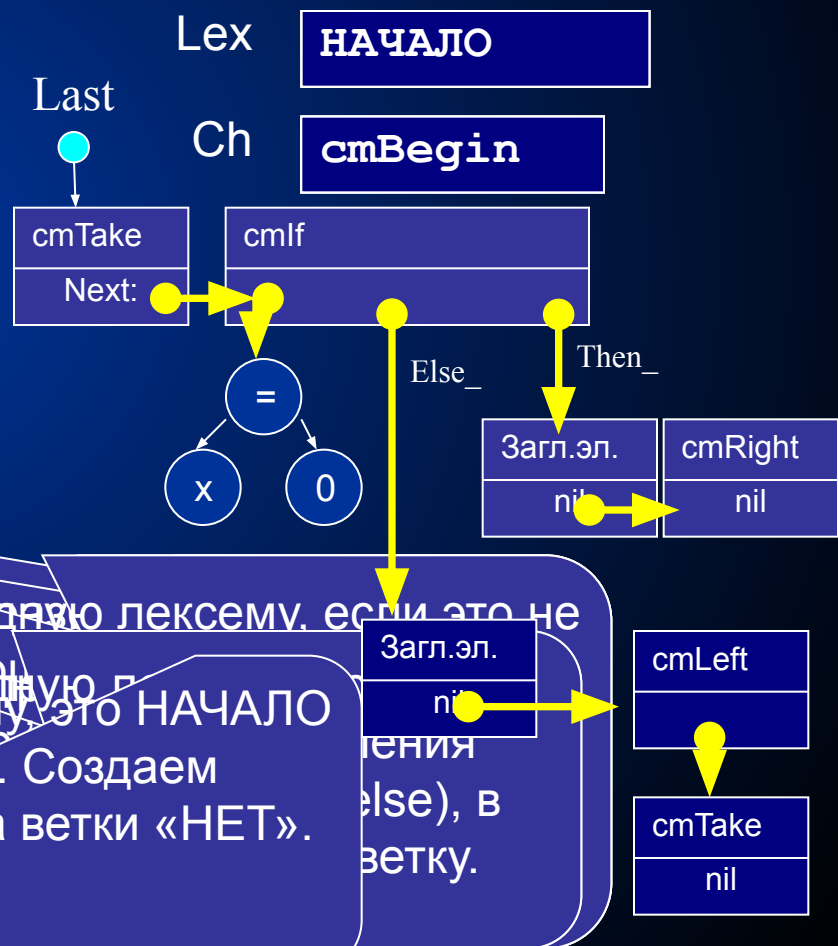
Поэтому введен новый тип:

```
pZnach=^tZnach;
tZnach=record
     p:integer;
     f:boolean;{true - число, false номер в массиве переменных}
end;
```

Если флаг f=true, то поле p – это переменная, которая хранит значение числа, если f=false, то p является индексом массива имен переменных,.

```
Procedure OperIf(var Last:pNode);
Var pp:pNode;
begin
   AddElem(Last,NewElem(cmIf));
   Uslovie(Last);
   Ch:=GetLex;
   if Ch<>cmThen
   then Error('Требуется «ТОГДА»');
   Ch:=GetLex;
   New(Last^.Then_);
   pp:=Last^.Then_;
   if Ch=cmBegin
   then Operator(pp)
   else SimpOper(pp)
   Ch:=GetLex;
   if Ch=cmElse
   then begin
          Ch:=GetLex;  New(Last^.Else_);
          pp:=Last^.Else_;
          if Ch=cmBegin then Operator(pp)
          else SimpOper(pp);
          Ch:=GetLex;
        end;
   if Ch<>cmEndIf
   then Error('Т
   Ch:=GetLex;
end;
```

```
ЕСЛИ X=0
ТОГДА ВПРАВО;
ИНАЧЕ НАЧАЛО
        ВЛЕВО;
        ВЗЯТЬ;
      КОНЕЦ
КОН_ЕСЛИ
```



Читаем очередную лексему, это НАЧАЛО – оператор составной. Создаем заглавный элемент списка ветки «НЕТ».

Интерпретатор объектного кода

- После того, как трансляция прошла успешно, и объектный код построен, можно приступать к исполнению программы. Для этого вызывается специальная подпрограмма, ей передается указатель на вход в список Code. Подпрограмма двигается указателем по списку, определяет тип каждого звена, выполняет команду, перемещая исполнителя (черепашку, ослика, зайчика, дятлика…) в зависимости от состояния рабочего поля. Приведу примитивный пример процедуры выводящей типы объектов списка.

```
procedure Interpetator(p:pNode);
begin
  While p<> nil do    {пока не конец списка}
   begin
    case p^.Typ of {в зависимости от типа звена}
     cmLeft,cmRight:write(MainLex[p^.Typ],'    ');
     cmIf: begin
           writeln(MainLex[p^.Typ]);     {вывести оператор ветвления}
           Write('Then:');
           Interpetator(p^.Then_);{разобрать ветку «Да»}
           Write('Else:');
           Interpetator(p^.Else_);{разобрать ветку «Нет»}
         end;
     cmFor: begin
             writeln(MainLex[p^.Typ]);
             Write('Тело Цикла');
             Interpetator(p^.Body);
             Writeln;
            end;
  e
end
```

В реальной программе в интерпретаторе не будет команд write, а при выполнении команд взаимодействия с полем (влево, вправо, взять, положить) будет вызываться специальная подпрограмма, которая анимационно будет изображать процесс перемещения (взаимодействия) на игровом поле. Другие команды (цикл, ветвление и т.д.) не будут отображаться на экране, но будут менять состояние переменных и списков

Конечные автоматы

Конечный автомат представляет собой особый способ описания алгоритма, который характеризуется набором из 5 элементов: K - конечный (ограниченный) набор состояний автомата, A - конечный алфавит, S - начальное состояние автомата, F - множество заключительных состояний автомата, D – отображение: откуда/куда.

Говорят, что конечный автомат допускает цепочку, если при ее анализе, начиная с начального состояния, функция D определена на каждом шаге и последнее состояние является заключительным.

Конечный автомат не допускает входную цепочку, если:

1) на каком-то шаге не определена функция D;

2) последнее состояние не является заключительным.

Пример. Конечный автомат, распознающий идентификатор.

K={0,1} множество состояний

A={' ', 'A'..'Z','a'..'z','0'..'9'} алфавит

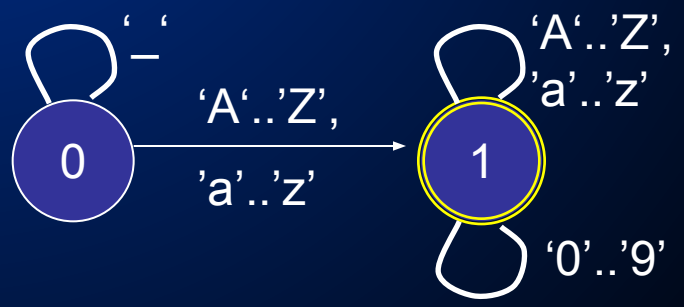
S=0 начальное состояние

F={1} конечное состояние

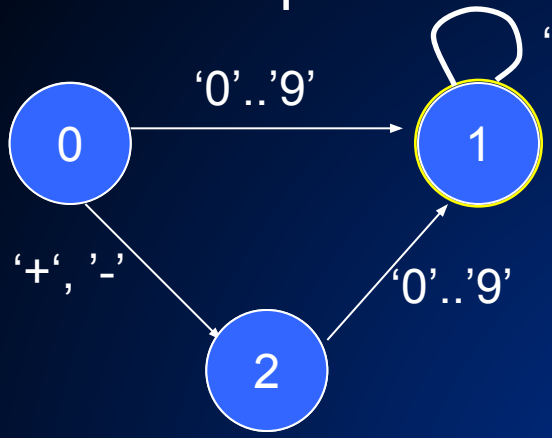
Конечный автомат можно задавать не только таблицей, но и диаграммой переходов.

D:

	входные символы		
	пробел	буква	цифра
0	0	1	Error
1	Error	1	1



- Описать конечный автомат, распознающий запись целого числа в десятичном виде.



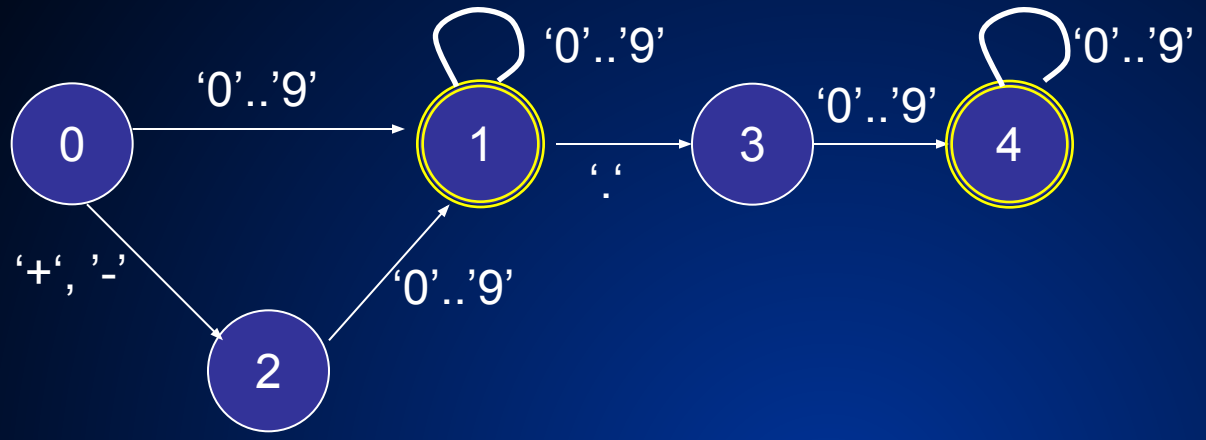
В чем проблема данного автомата?

Посмотрим, как работает автомат для числа s='-15'

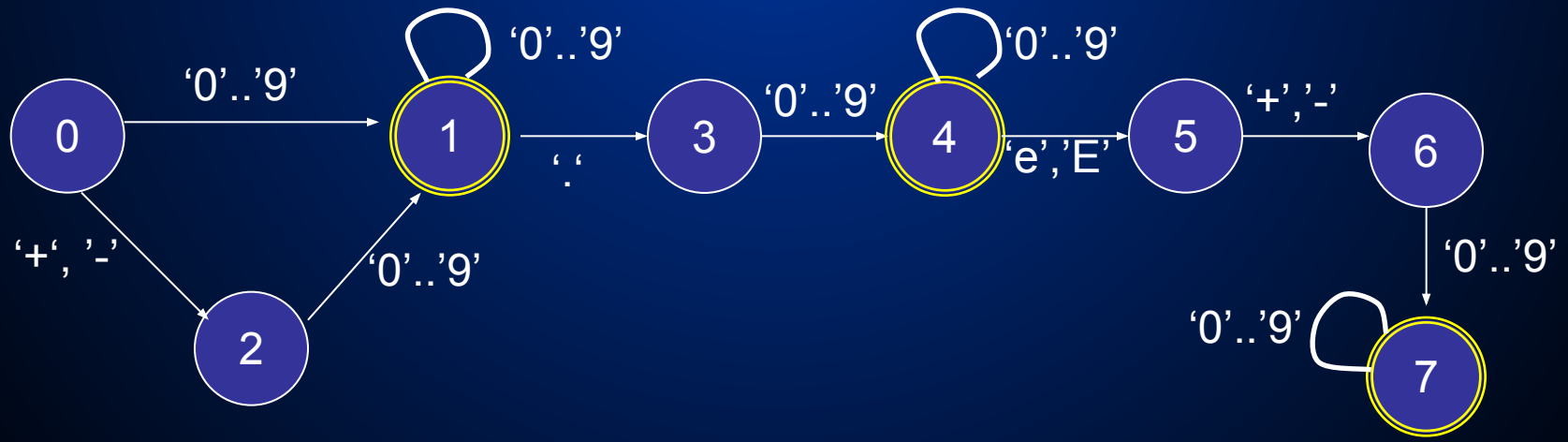
```
Function IntA(s:string):boolean;
Var i, q :integer; f:boolean;
Begin
   q:=0; i:=0; f:=true;
   repeat
    inc(i);
    case q of
     0: if s[i] in ['0'..'9']
        then q:=1
        else if s[i] in ['+','-']
             then q:=2 else f:=false;
       2: if s[i] in ['0'..'9']
           then q:=1 else f:=false;
       1: if not (s[i] in ['0'..'9'])
           then f:=false
        end
   until (not f)or(i>=length(s));
   IntA:=f and (q=1)
End;
```

S	-15
q	1

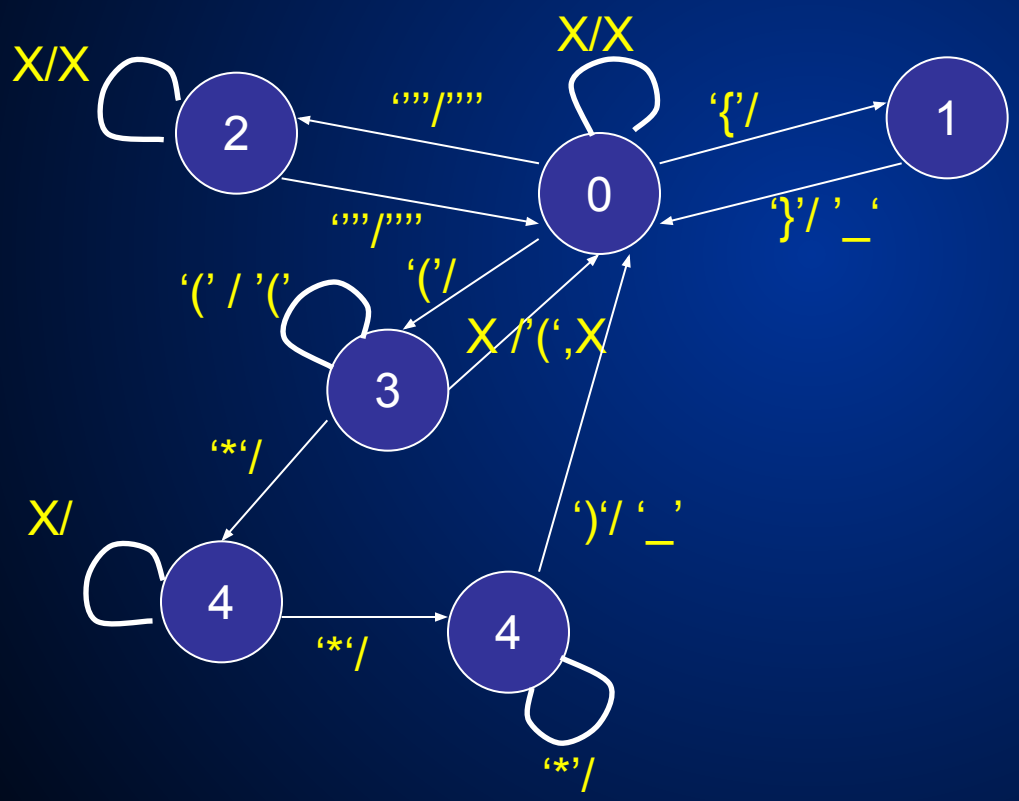
- Описать конечный автомат, распознающий запись дробного числа в десятичном виде.



- Описать конечный автомат, распознающий запись дробного числа типа real.



- Дана работоспособная программа на Паскале. Необходимо удалить из нее комментарии так, чтобы сохранить ее функции:
- А) разрешается использовать комментарии только одного типа. Все, что заключено в фигурные скобки ‘{‘, ‘}’ считается комментарием. Комментарии не могут быть вложены друг в друга.
- Б) разрешается использовать комментарии только двух типов. Все, что заключено в фигурные скобки ‘{‘, ‘}’ или (*. *) считается комментарием. Комментарии разного типа могут быть вложены друг в друга.



Есть ли еще проблемы? За каждую найденную - +5 к краме ☺

Описание языков с использованием графов

- Опишем наш уже созданный язык, но в терминах графа. Каждой дуге будет приписана лексема, переводящая автомат в другое состояние, а также некоторой действие со стеком. В стеке мы будем сохранять состояние, в которое необходимо вернуться, после завершения анализа некоторого оператора.

Самая сложная для понимания часть – анализ операторов. Дело в том, что операторы начинаются после слова «НАЧАЛО» (после начала программы). Однако, некоторые операторы, например оператор ветвления, могут в свою очередь содержать другие операторы, в том числе и оператор ветвления и т.д. Чтобы не увеличивать число состояний автомата, усложнять его вид, было принято решение об организации специального стека состояний. Перед началом анализа оператора, автомат заносит в этот стек номер состояния, в которое нужно вернуться после завершения анализа данного оператора. Когда анализ оператора завершен, автомат выполняет команду Sost:=Pop - извлечь из стояние, в которое необходимо вернуться.

Если после описания типа встретилась лексема «НАЧАЛО», то переходим в состояние 7 (анализ тела программы), если нет, то это описание следующей группы переменных. Например,

X, У: ЦЕЛЫЕ;

A, B: ДРОБНЫЕ;

```
Procedure Automat (sost : integer);
Var   Ch: Ident;
…;
Begin
  Ch:=GetLex; {Получить первую лексе
  repeat
    Case Sost of
      0:if Ch<>cmProgram
       then Error ('Требуется слово «Программа»')
       else begin
              Sost:=1;Ch:=GetLex
            End;
      1:if Ch<>cmIdent
        then Error ('Требуется имя программы')
        else begin
               Ch:=GetLex
               If Ch<>cmTZ {";"}
               Then Error('Требу
               Sost:=2;Ch:=GetLe
          End;
```

Находясь в состоянии 1, автомат ждет идентификатор имени программы, если его нет, то выводим сообщение об ошибке, иначе ищем «;» и переходим с состояние 2.

Находясь в состоянии 0, автомат ждет слово ПРОГРАММА, если его нет, то выводим сообщение об ошибке, иначе переходим с состояние 1.

```
2: begin
   if Ch=cmIdent {если имя пер
   then sost:=3
    else   if Ch=cmBegin  {
         then begin
                Sost:=7;Push(Stop)  {занести в сте  мер конечного состояния}
              end
         else Error ('Требуется «НАЧАЛО»')
   end;
3:begin запомнить имя переменной; Ch:=GetLex;
       If Ch=cmTT {":"}  Then Sost:=5
       Else if Ch=cmZP {","}    Then Sost:=4
         Else Error('Требуется "," или ":"')
   end;
4: begin Ch:=GetLex;
      if Ch=cmIdent {если имя переменной, то} then sost:=3
      else Error ('Требуется имя переменной')
   end;
5: begin    {если ":"}
     Ch:=GetLex; If not (Ch in [cmInt, cmReal, cmChar])
                 Then Error ('
                 Else Sost:=6;
   end;
```

Находясь в состоянии 5, автомат ждет имя типа , если его нет, то выводим «ошибка», иначе переходим в состояние 6

Находясь в состоянии 3, автомат может встретить «:», тогда перечисление имен переменных закончилось, надо ждать имя типа (состояние 5) или «,» - ждем имя переменной (состояние 4)

```
6:begin
   заполнить та
   начальное зна
   Ch:=GetLex;
   If Ch=cmBegi
   Then begin
          Sost:
          Push(S
        end
   Else if Sost=cmIdent Then Sost:=3
     Else Error ('Требуется «НАЧАЛО»')
   End;
7: begin
    Ch:=GetLex;
    If Ch=cmEnd Then Sost:=20
    Else if cm in [cmLeft, cmRight, …]
         Then begin
                Sost:=9;  Push (19)
              end
         Else Erro
  end;
```

В состоянии 7 мы можем встретить лексему КОНЕЦ, что означает завершение логического блока – переходим в состояние 20, или можем встретить любой оператор: ВЛЕВО, ЕСЛИ, цикл и т.д. В этом случае переходим в состояние 20 для их разбора. Не забываем запомнить состояние, куда надо вернуться (Push (19)).

В состоянии 6 закончилось описание списка переменных, заносим их в таблицу имен и читаем следующую лексему. Если это НАЧАЛО, то переходим в состояние 7 и разбираем операторы, если – идентификатор, то разбираем новый блок описания переменных (состояние 3)

```
 19:begin {оператор закончился}
  if Ch<>cmTZ  {";"}
  then Error ('Требуется ";"')
  else Sost:=7
   end
 20: Sost:=pop;
 9:begin
    {сюда необходимо вставить блок кода для анализа
  операторов}
  end;
 End; {case}
Until Sost=Stop {завершить анализ при
  достижении конечного состояния}
End;
```

{блок кода для состояния 9, рассматривается отдельно, чтобы не загромождать основную процедуру}

```
Case  Ch of
  CmLeft, cmRight, cmUp, cmDown: Begin
    Создать объектный код данных операторов;
    Sost:=POP;   {извлечь состояние, в которое нужно вернуться}
    Ch:=GetLex
   End;
  CmIf:begin
     Анализ оператора ветвления:
     Case Sost of
       …
     End;
        Sost:=POP; Ch:=GetLex
   End;
   …
 End; {case}
End;
```

- Представленная процедура позволяет реализовать анализ конкретного языка, но можно пойти дальше, разработать структуру данных, позволяющую представлять исходный текст программы в виде графа состояний автомата. Каждой дуге может быть приписана лексема, появление которой переводит автомат в следующее состояние, а так же действие, которое необходимо при этом выполнить. Например, сохранить текущее состояние в стеке, извлечь состояние, в которое необходимо вернуться и т.д. Создать такой универсальный автомат весьма непросто, но мы и не ставим такой цели, кого это заинтересовало, то может сделать.

Пошаговая трассировка. Определение точки возникновения синтаксической ошибки.

- В предыдущей части мы рассмотрели процесс анализа исходного текста программы, ее перевода на объектный язык. В процессе такого анализа синтаксический анализатор может обнаружить какое-либо несоответствие или синтаксическую ошибку. У нас это выглядело так:
- If Ch<> чему-то
- Then Error('Требуется то-то');
- Понятно, что процедура Error выводит сообщение «Требуется то-то», но пользователю очень сложно понять, в какой части программы требуется «ТО-ТО». Хочется, чтобы синтаксический анализатор не только сообщал об ошибке, но и показывал точку в тексте программы, где эта ошибка была обнаружена. Чтобы решить эту проблему, достаточно в каждом звене объектного кода хранить ссылку на строку текстового редактора с ее текстовым эквивалентом. Ссылка может быть представлена в виде пары чисел: номер строки, номер слова в строке. Такой подход позволит не только указать место возникновения синтаксической ошибки на этапе компиляции, но и реализовать пошаговую трассировку программы (подсвечивая строку с исполняемой командой), устанавливать точки останова.

Противники

- Игра становится гораздо интереснее, если выполнению миссии нашего героя (черепашки, кота, зайца) мешают противники. Например, Вини Пуху собирать мед могут мешать пчелы, Красной Шапочке – серые волки и так далее. Самый простой вариант – это автоматическое движение противника по горизонтали до непроходимого препятствия или границы поля, наткнувшись на препятствие, противник разворачивается и движется в противоположном направлении. Встреча исполнителя и противника в одной зоне экрана означает завершение игры и проигрыш исполнителя, а для пользователя – исправлять программу. Более интеллектуальный вариант – связать противника с программой управления, которую тоже может написать пользователь. В этом случае мы будем иметь дело с соревнованием программ и многозадачным интерпретатором, так как ему придется одновременно выполнять несколько программ.

Многозадачный транслятор

- В отличие от однозадачного транслятора, многозадачный имеет не один динамический список с объектным кодом, а несколько, каждый список соответствует одной из нескольких программ. Возникает масса вопросов и проблем:
- Когда надо прекратить исполнение текущей программы и перейти к следующей?
- Куда возвращаться? (на какое звено объектного кода)
- Как обеспечить взаимную независимость программ и их переменных?

Список программ –

Массив Task

```
Repeat
  for i:=0 to NumbGamer-1
    Interpretation(Task[i]);
  нарисовать изменения на поле
Until игра завершена
```

0

Игрок 0

Вход в объектный код

Текущая команда

Переменные

A	B	X	…
0	5	1	

1

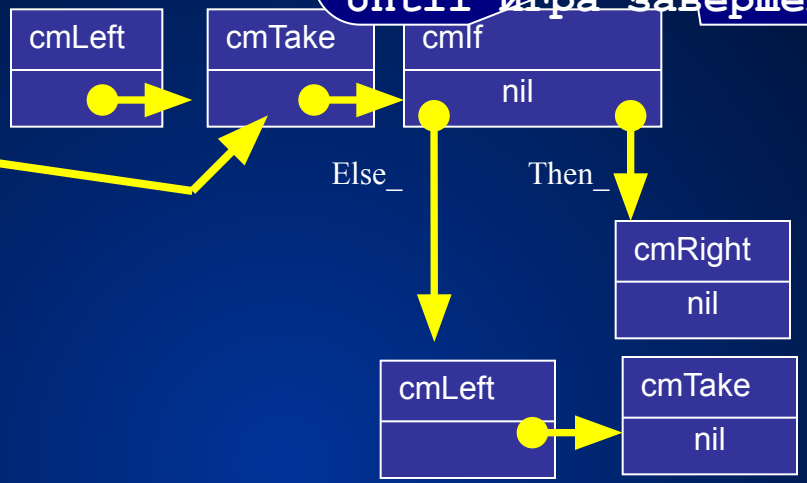
Игрок 1

Вход в объектный код

Текущая команда

A	B	X	…
0	5	1	

…



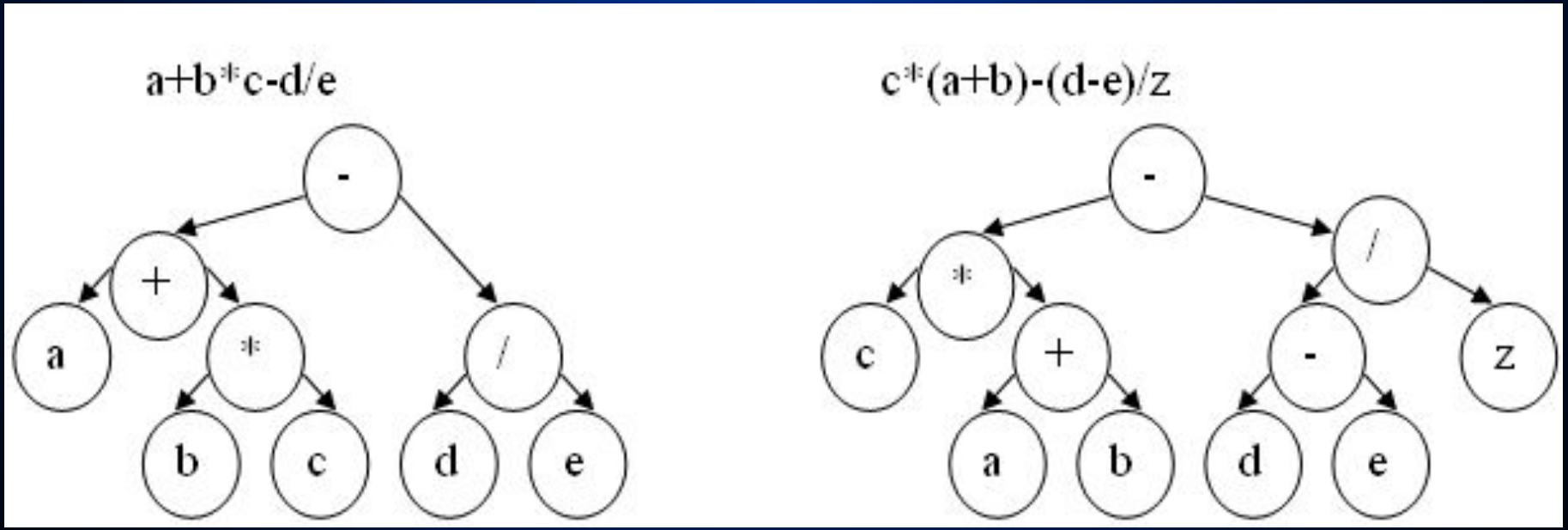
Как только интерпретатор встретил команду перемещения исполнителя, он помещает в очередь произошедшие события и прерывает исполнение текущей программы, начинает следующую. Когда все программы отработали необходимо анимационно изобразить все события на поле.

Представление сложных условий в операторе ветвления. Оператор присваивания

- При исполнении объектного кода возникает проблема хранения сложных условий в операторе ветвления. Проблема может быть решена простым запретом, но это слишком простой и абсолютно не эффективный путь.
- Если X<8 тогда … {простое условие}
- Если (X<8)И(X>20)ИЛИ(Z=0) тогда … {сложное логическое выражение}
- Представление простых условий мы уже рассмотрели раньше. Теперь рассмотрим идею представления сложных, но прежде вернемся к оператору присваивания. Слева от оператора присваивания может стоять только переменная, а вот справа любое арифметическое выражения, причем в нем могут встречаться не только четыре арифметических действия, но и скобки, меняющие порядок действий.
- Например, x:=(2+y)*34-(y+z)/4. Порядок действий в таких выражениях вовсе не очевиден. Существует классический алгоритм, позволяющий перевести выражение в особый вид – обратную польскую запись. Но работать с таким представлением не всегда удобно. Воспользуемся древовидным представлением.

Разбор арифметических выражений, содержащих скобки

- В некоторых задачах необходимо вычислить значение арифметического выражения, возможно содержащего скобки. Его очень просто вычислить, если суметь преобразовать арифметическое выражение в древовидную структуру такого типа:



- Разделим эту задачу на две части: разбор и строительство дерева по арифметическому выражению без скобок, работа со скобочной структурой.
- **Разбор выражения без скобок**. Просматривается арифметическое выражение слева направо, ищутся высокоприоритетные операции ‘*’, ‘/’. У найденной операции выделяются левый и правый операнд. Строится дерево из трех узлов. Данная структура помещается в особый массив, а операция и операнды заменяются особой переменной, к которой прикрепляется построенная структура.

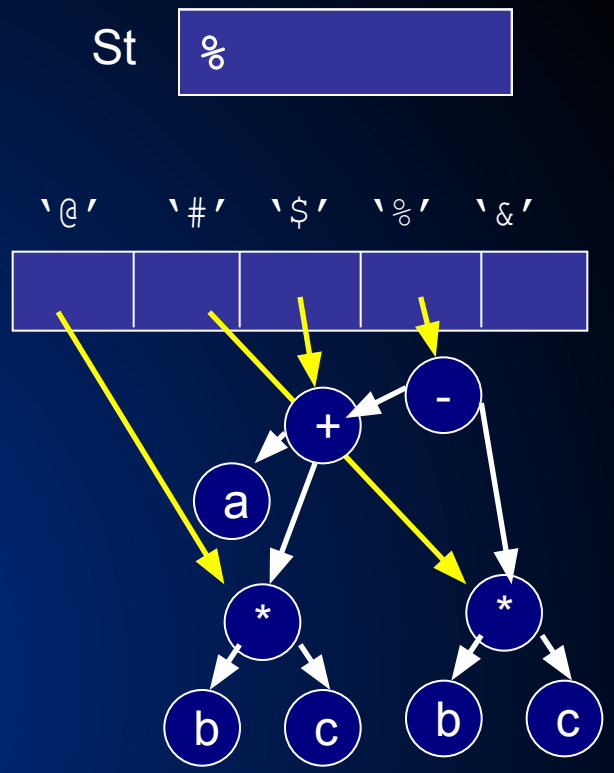
Затем, когда все высокоприоритетные операции закончатся, проводим ту же операцию над сложением и вычитанием. Если один из операндов, спец. переменная, то «цепляем» уже созданный узел из массива Mass. Рассмотрим основные типы:

```
Type Ref=^Node;
     Node=record
            Lit:char;
            L,R:Ref;{Массив переменных}
          end;
    alf='A'..'Я';
 var
    Num:Alf;{количество переменных}
    Mass:array[Alf] of ref;
    Root:ref; {корень дерева}

Function NewEl(c:char):Ref;
{Создает новое звено, со значением С}
var tz:ref;
begin
 New(tz); tz^.L:=nil;tz^.R:=nil; tz^.Lit:=c;   NewEl:=tz;
end;
```

```
procedure Arif(m:SetC); {Ищет только операции из множества m}
var i:integer;     tz:ref;     f:boolean;
begin
   f:=false;
   Repeat
     i:=1; {Ищем только операции из множества m}
     While (not (st[i] in m)and(i<length(st))) do
       inc(i);
     if st[i] in m  {если операция найдена, то}
     then begin
          tz:=NewEl(st[i]); {создать новое звено}
          if st[i-1] in ['a'..'z'] {если найден операнд, то}
          then tz^.L:=NewEl(st[i-1]) {создать новое звено, прицепить его слева}
          else tz^.L:=mass[st[i-1]];{если это замененная переменная, то прицепить ее}
          if st[i+1] in ['a'..'z'] {аналогично справа…}
          then tz^.R:=NewEl(st[i+1])
          else tz^.R:=mass[st[i+1]];
          Delete(st,i-1,2); {удалить операцию и один операнд}
          inc(Num); Mass[Num]:=tz; {создать новую переменную в массиве}
          st[i-1]:=Num
        end
     else f:=true{если операций уже не осталось, то закончить цикл}
   until f;  end;
```

```
procedure Arif(m:SetC);
var i:integer;     tz:ref;    f:boolean;
begin
 f:=false;
 Repeat
  i:=1; {Ищем только операции из множества m}
  While (not(st[i] in m)and(i<length(st))) do
   inc(i);
  if st[i] in m  {если операция найдена, то}
  then begin
         tz:=NewEl(st[i]);
         if st[i-1] in ['a'..'z']
         then tz^.L:=NewEl(st[i-1])
         else tz^.L:=mass[st[i-1]]
         if st[i+1] in ['a'..'z']
         then tz^.R:=NewEl(st[i+1])
         else tz^.R:=mass[st[i+1]];
         Delete(st,i-1,2);
         inc(Num); Mass[Num]:=tz;
         st[i-1]:=Num
       end
     else f:=true
   until f;
end;
```



Повторяем для вычитания, создаем звено и помещаем его в массив. Теперь mass[%] хранит вход в дерево арифметического выражения

```
Function Calc(St:string):Ref; {Строит дерево по арифметическому выражению без скобок}
begin{Calc}
   Arif(['*','/']);{обработать умнож. и деление}
   Arif(['+','-']); {теперь сложение и вычитание}
   Calc:=mass[st[1]]
end;
```

Скобочные выражения анализируются так: ищется самая левая закрывающая скобка ‘)’, затем ее открывающая пара. Выражение между этих скобок вырезается из строки и разбирается функцией Calc.

```
Function CalcScob(st:string):ref;
var f:boolean;      i,j:integer;    tz:ref;
begin
  f:=false;CalcScob:=nil;
  Repeat
    i:=Pos( ')', st); {Ищем самую левую скобку ‘)’}
    if  i<>0     then begin{если нашли, то}
       j:=i;
       While (st[j]<>'(')and(j>0) do dec(j); {ищем ее пару}
       if j=0 then exit; {если пары нет, то ошибка}
       tz:=Calc(copy(st,j+1,i-j-1));{разбираем выражение между скобок}
       delete(st,j,i-j); {удаляем из строки разобранное выражение}
       inc(num); mass[Num]:=tz; st[j]:=Num; {создаем переменную}
         end
    else f:=true
  until f;
  if length(st)<>1    then CalcScob:=Calc(st)
  else CalcScob:=mass[st[1]];
end;
```

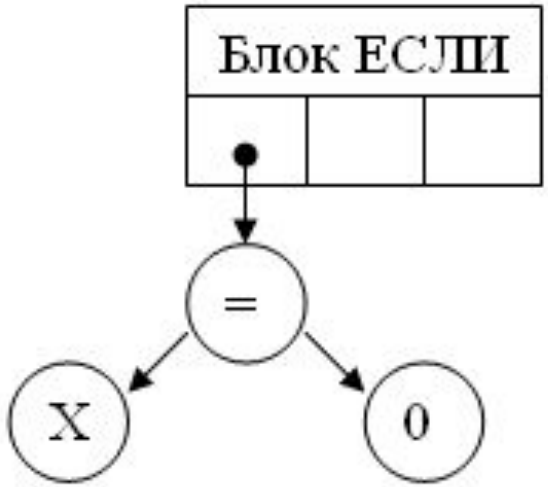
- Дальнейшее развитие этой программы очевидно – переменные a, b, c, … можно заменить числами или создать массив значений переменных:
- Var Value:array[‘a’..’z’] of real;
- Каждый элемент массива Value имеет литерный номер и вещественное значение. Чтобы получить значение некоторой переменной нужно: Пусть tz^.lit=’a’, тогда Value[tz^.lit]=значение переменной. Рассмотрим функцию нахождения результата арифметического выражения представленного в дереве, построенном предыдущей программой.

```
Function Arifmet(Root:ref):real;
Begin
 If Root^.L=Root^.R {Если это лист, то Root^.L=Root^.R = nil}
 Then Arifmet:=Value[Root^.Lit]
 Else case Root^.Lit of
      '+': Arifmet:= Arifmet (Root^.L)+ Arifmet(Root^.R);
      '-': Arifmet:= Arifmet (Root^.L) - Arifmet(Root^.R);
      '*': Arifmet:= Arifmet (Root^.L) * Arifmet(Root^.R);
      '/': Arifmet:= Arifmet (Root^.L) / Arifmet(Root^.R);
     end;{case}
End;
```

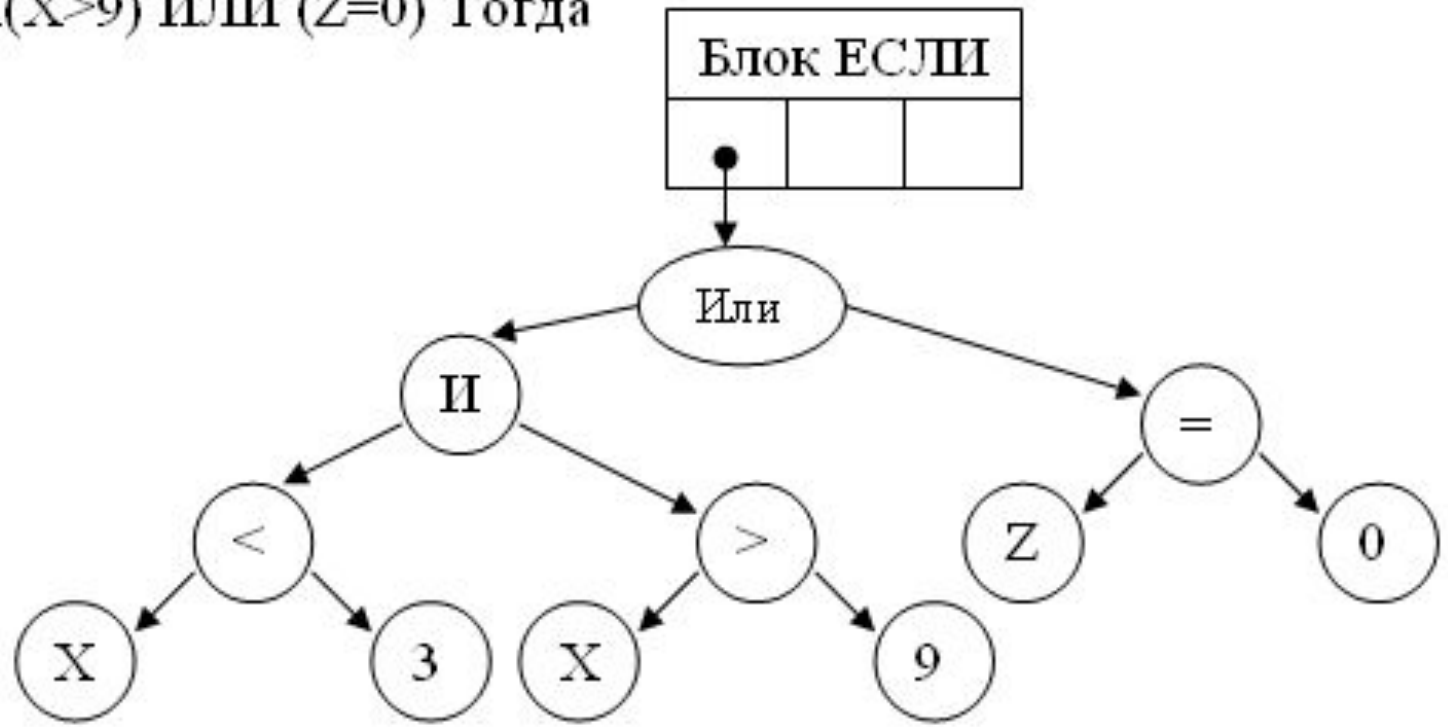
Для простоты мы не анализировали случай пустого дерева, когда Root=nil.

Вернемся к логическим выражениям. Любое логическое условие преобразуем в древовидное представление.

Если X=0 тогда



Если (X<3)И(X>9) ИЛИ (Z=0) Тогда



- Звено, которое представляет оператор ветвления в объектном коде, имеет указатель на древовидную структуру. Именно она хранит логическое выражение, истинность которого и определяет ветку дальнейшего движения. Вычислить результат довольно просто. Каждый лист такого дерева хранит либо числовое значение, либо индекс переменной в массиве переменных. Все не «листы» хранят либо логическую операцию («И», «ИЛИ»), либо операцию сравнения (‘<’,‘>’,‘=’,‘<>’,‘<=’,‘>=’).

```
Type pRef=^tNode;
tNode=record
      left, right:pRef;
      case TypeNode:byte of
      cmNumber: Num:real; {хранится число}
      cmVariable: Index: integer; {хранится индекс переменной}
      cmOperation: Oper:string[3];{или char, операция}
   End;
```

- В момент компиляции происходит перевод арифметических и логических выражений в древовидное представление. В момент исполнения объектного кода, интерпретатор вычисляет значение выражения и использует его по назначению.

Разбор арифметических выражений методом рекурсивного спуска

Для арифметического выражения *дерево разбора* можно описать так. В корне бинарного дерева находится знак операции, которая должна быть выполнена последней. Левым поддеревом (сыном) является дерево, представляющее первый из операндов последней операции, правым — второй из операндов. Если эта операция унарная, то левое поддерево отсутствует.

Подсчет значения арифметического выражения, основанный на таком способе представления рекурсивен и фактически совпадает с рекурсивным определением самого выражения.

<выражение>::=<терм>|<терм>+<выражение>|<терм>-<выражение>

<терм>::=<множитель>|<множитель>*<терм>|<множитель>/<терм>

<множитель>::=(<выражение>)|<имя>|<натуральное число>

<имя>::=<буква>|<имя><буква>|<имя><цифра>

<натуральное число>::=<цифра>|<натуральное число><цифра>

<цифра>::=0|1|2|3|4|5|6|7|8|9

<буква>::=_|A|B|…|Z|a|b|…|z|

Так, выражение представляет собой сумму слагаемых (здесь под суммой подразумевается выполнение операций, как сложения, так и вычитания). Каждое слагаемое представляет собой произведение множителей (сюда же отнесена операция деления). А множитель представляет собой либо число, либо переменную, либо выражение в скобках (при наличии унарного минуса множителем является также выражение со стоящим перед ним знаком минус).

```
var s:string;{исходное выражение}
    i:integer;{номер текущего символа}
function Mul:longint; Forward;
function Factor:longint; Forward;

function Add:longint; {суммирует слагаемые}
var q,res:longint;    c:char;
Begin
   res:=Mul;{первое слагаемое}
   While s[i] in ['+','-'] do
    Begin
     c:=s[i];  i:=i+1;  q:=Mul;{очередное слагаемое}
     case c of
        '+':res:=res+q;
        '-':res:=res-q;
      End
   End;{While}
  Add:=res
End;
```

```
function Mul:longint; {перемножает множители}
var q,res:longint;      c:char;
Begin
   res:=Factor;{первый множитель}
   While s[i] in ['*','/'] do
    Begin
     c:=s[i]; i:=i+1;q:=Factor;{очередной множитель}
     case c of
      '*':res:=res*q;
      '/':If q=0
          then Begin writeln('деление на 0');
                     halt
               End
          else res:=res div q
      End {case}
    End; {While}
   Mul:=res
End;
```

```
function Number:longint;{выделяет число}
var res:longint;
Begin   res:=0;
 While (i<=length(s)) and (s[i] in ['0'..'9']) do Begin
       res:=res*10+(ord(s[i])-ord('0')); i:=i+1
  End; {While}
 Number:=res
End;
function Factor:longint; {выделяет множитель}
var q:longint;      c:char;
Begin
  case s[i] of
   '0'..'9':Factor:=Number;
   '(':Begin  i:=i+1;Factor:=Add; i:=i+1;{пропустили  ')'}End;
   '-':Begin i:=i+1; Factor:=-Factor;   End
   else Begin writeln('ошибка');  halt    End
  End {case}
End;

Begin {основная программа}
 readln(s); i:=1;  writeln(Add)
End.
```

- Арифметическое выражение представляет собой сумму нескольких слагаемых (возможно одно). Каждое слагаемое это произведение множителей (возможно одно). Каждый множитель это либо число, либо выражение в скобках. А выражение в скобках – это сумма слагаемых и так далее. Процедура Add выделяет первое слагаемое (Mul) – это либо число, либо произведение, которое надо сосчитать раньше, либо выражение в скобках. За выбор слагаемого отвечает процедура Mul, которая выделяет первый множитель (Factor).

```
function Add:longint; {суммирует слагаемые}
Begin
   res:=Mul;{первое слагаемое}
   While s[i] in ['+','-'] do Begin
     c:=s[i];  i:=i+1;  q:=Mul;{очередное слагаемое}
     case c of
        '+':res:=res+q;
        '-':res:=res-q;
      End
   End;{While}
  Add:=res
End;
function Mul:longint; {перемножает множители}
Begin
  res:=Factor;{первый множитель}
  While s[i] in ['*','/'] do Begin
     c:=s[i]; i:=i+1;q:=Factor;{очередной множитель}
     case c of
      '*': res:=res*q;
      '/': res:=res div q
      End {case}
    End; {While}
   Mul:=res
End;
function Number:longint;{выделяет число}
Begin
End;
function Factor:longint; {выделяет множитель}
Begin
  case s[i] of
   '0'..'9':Factor:=Number;
   '(':Begin  i:=i+1;Factor:=Add; i:=i+1; End;
   '-':Begin i:=i+1; Factor:=-Factor;   End
   else Begin writeln('ошибка');  halt    End
  End {case}
End;
```

S `5*2+7*3-11`

Результат `20`

Прибавляем слагаемое 21, сдвигаемся по строке, обнаруживаем операцию '-', вызываем Mul, которая при помощи factor обнаружит число 11 и вернет его нам, остается найти результат.

Динамический линейный однонаправленный список

Основными проблемами классического статического списка на основе массива являются:

- неэффективное распределение памяти (резервирование лишней);
- невозможность увеличить размер списка в ходе программы;
- медленные операции вставки и удаления элементов без нарушения порядка их следования (сдвиг элементов в среднем O(n/2) штук).
- От этих проблем избавлен динамический список. Он формируется по мере необходимости путем добавления (вставки) нового звена. Рассмотрим основные операции над динамическим списком:

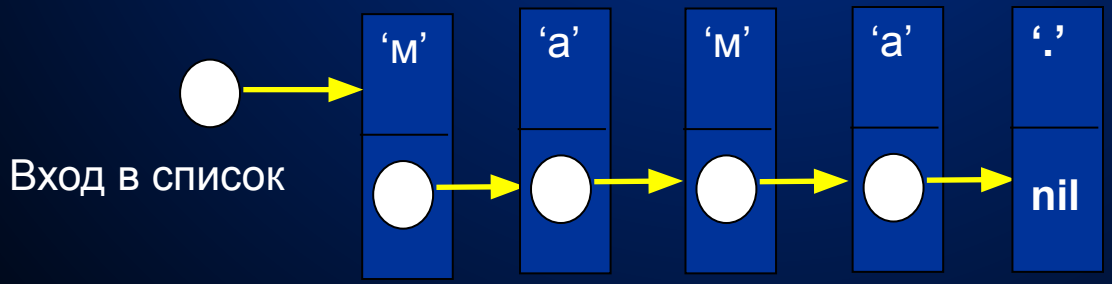
Формирование списка

Рассмотрим структуру данных. Для примера в списке будем хранить литеры (буквы), причем признаком конца ввода выберем символ ‘.’, это условность, формально можно взять любой другой символ.

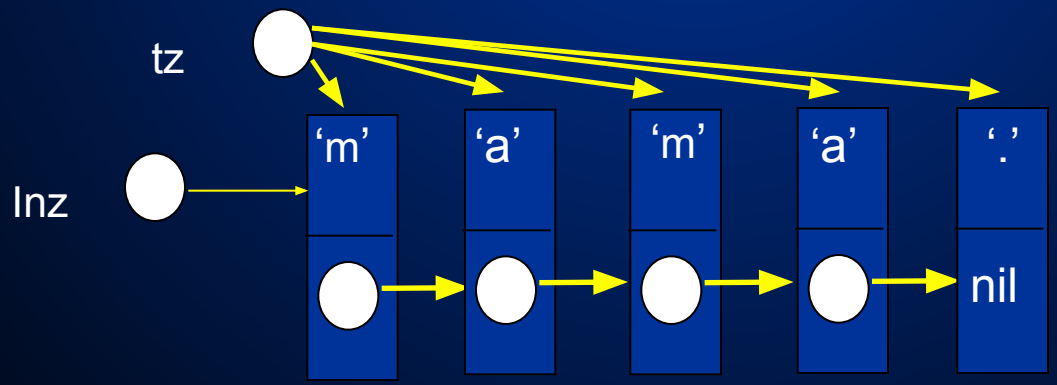
```
Type    ref=^Node;{указатель на звено}
     Node=record{звено}
          Next:Ref; {указатель на следующее звено}
          Lit:char {информация звена (символ)}
     End;
```

Каждое звено хранит один информационный символ и указатель на следующее звено.

Сам список будет выглядеть так:



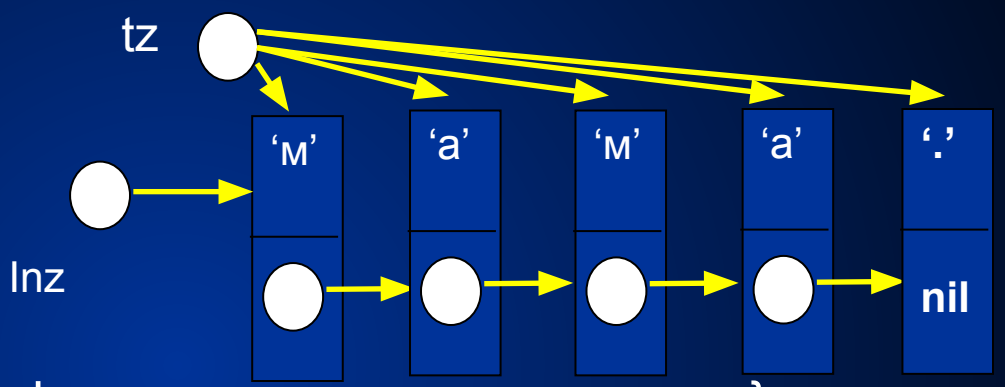
```
Procedure CreateList(var inz:ref);
Var tz:ref; a:char;
Begin
   New(inz); tz:=Inz;
   Read(a);tz^.Lit:=a;
  {дополнительный указатель tz потребовался, так как inz
    должен хранить адрес первого звена}
   While a<>'.' Do
     Begin
        New(tz^.next);{создадим новое звено}
        Tz:=tz^.Next;{перейдем к следующему звену}
        Read(a); tz^.lit:=a
     End;
   Tz^.Next:=nil; ;{ пометим конец списка}
   Readln ;{ удалим enter из буфера клавиатуры}
End;
```



•**Вывод списка.**

Встаем на начало списка (inz), двигаемся по нему, переходя к следующему элементу (поле Next) и выводим информацию (поле lit).

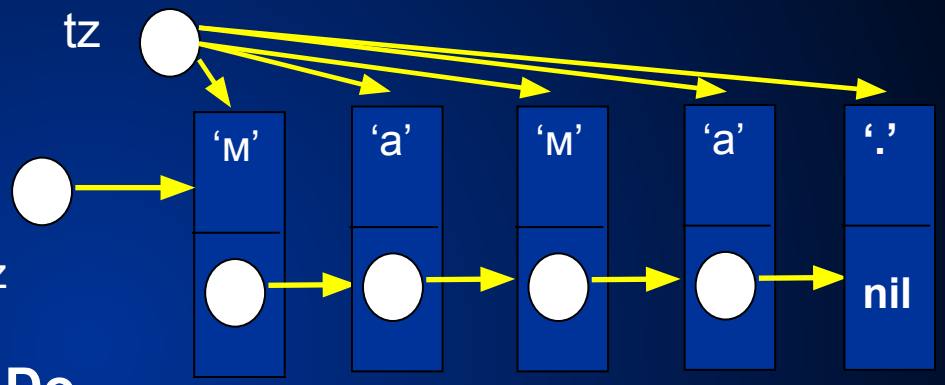
```
Procedure WriteList(inz:ref);
Var tz:ref;
Begin
   tz:=Inz;
   While tz<>nil Do
      Begin
        write(tz^.Lit);{выведем информацию из текущего звена}
        Tz:=tz^.Next;{перейдем к следующему звену}
      End;
End;
```



Признаком конца списка мы используем не символ ‘.’, а пустую ссылку nil. Это позволяет оторваться от конкретного списка и создать универсальную процедуру вывода списка, которая не зависит от того, что в нем содержится.

•Поиск элемента в списке.

Одна из важнейших операций в программировании. Встаем на начало списка (inz), двигаемся по нему, переходя к следующему элементу (поле Next), пока не найдем искомый элемент или пока не дойдем до конца списка (nil).




```
function Seek(inz:ref;key:lit):ref;
Var tz:ref;
Begin
   tz:=Inz;
   While (tz<>nil)and(tz^.Lit<>key) Do
        Tz:=tz^.Next;{перейдем к следующему звену}
   Seek:=tz
End;
```

Мы имеем два сравнения на каждый элемент, поэтому худшая скорость будет O(2n), а средняя – O(2n/2)=O(n). Обратите внимание, что в цикле while используется логическая операции and, а не or! Это условие продолжения, а не окончания!

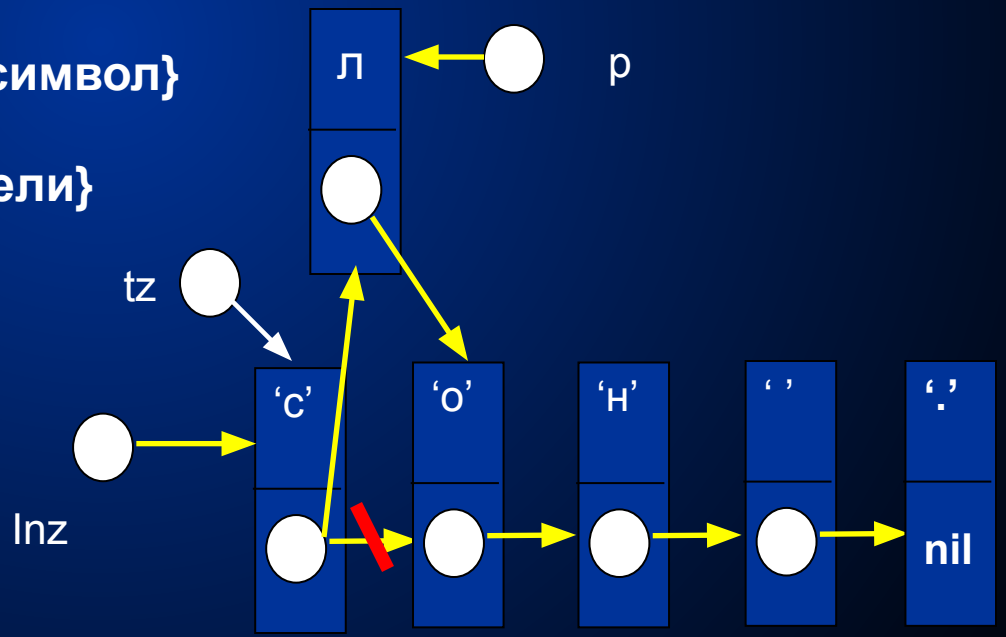
•Вставка элемента в список.

В отличие от статического списка, в котором для вставки элемента необходимо сместить все правые элементы на одну ячейку вправо, что требует в среднем порядка O(n/2) операций копирования, в динамическом – вставка осуществляется за O(1). Существуют два варианта вставки:

а) <u>вставка после текущего</u>.

Пусть tz указывает на некоторый элемент списка, необходимо за ним разместить новое звено с заданным символом.

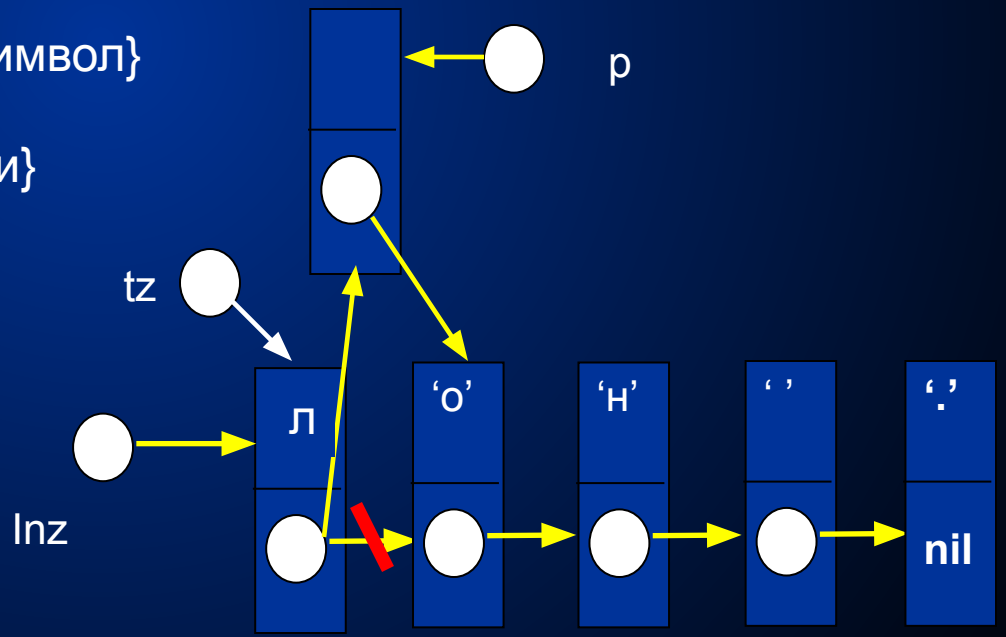
```
Procedure InsAfter(tz:ref;a:char);
Var p:ref;
Begin
   New(p);{создадим новое звено}
   P^.lit:=a; {запомним заданный символ}
   P^.Next:=tz^.next;
   Tz^.Next:=p; {перекинем указатели}
End;
```



б) вставка перед текущим.

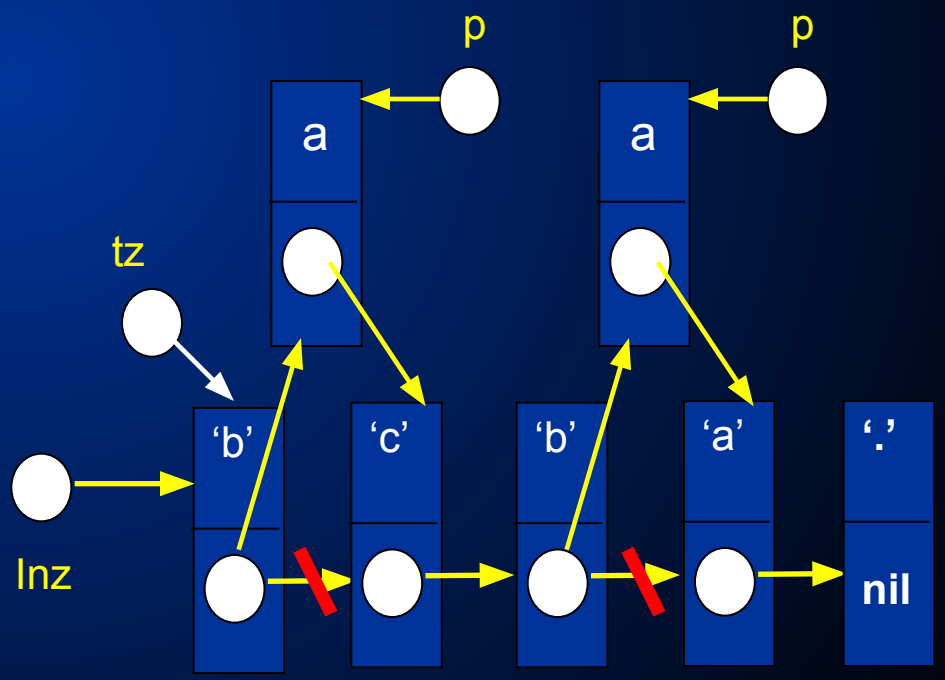
Пусть tz указывает на некоторый элемент списка, перед которым необходимо разместить новое звено с заданным символом. Кажется, что это невозможно, так как мы не имеем доступа к предыдущим элементам и, следовательно, не сможем изменить связи в цепочке. Однако, если нельзя, но очень надо, то можно! ☺ Для этого вставим новое звено после текущего, а значение текущего звена скопируем во вновь созданное. В результате вставляемый символ окажется перед символом, который был текущим, то есть задача выполнена!

```
Procedure InsBefore(tz:ref;a:char);
Var p:ref;
Begin
   New(p);{создадим новое звено}
   P^.lit:=tz^.lit;{запомним текущий символ}
   P^.Next:=tz^.next;
   Tz^.Next:=p; {перекинем указатели}
   Tz^.Lit:=a
End;
```



Дан линейный динамический список, вставить в нем букву ‘a’ после буквы ‘b’.

```
Procedure InsAfter(tz:ref;a:char);
Var p:ref;
Begin
    New(p);
    P^.lit:=a;
    P^.Next:=tz^.next;
    Tz^.Next:=p;
End;
Procedure InsA(var inz:Ref;
                a,b:char);
Vat tz:Ref;
Begin
  tz:=Inz;
  While tz<>nil do begin
    if tz^.Lit=b
    then InsAfter(tz,a);
    tz:=tz^.Next
  end
End;
```



Удаление элемента из списка.

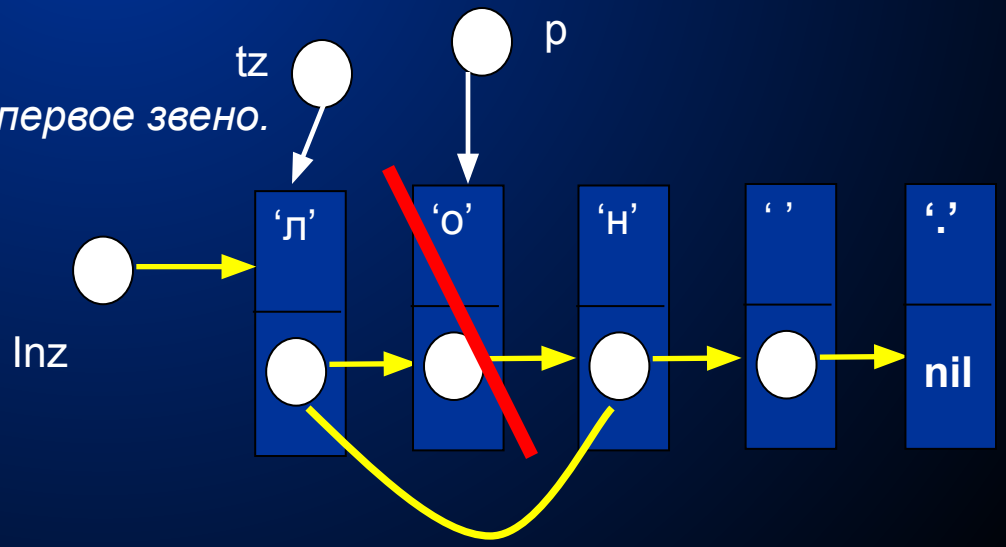
В отличие от статического списка, в котором для удаление элемента необходимо сместить все правые элементы на одну ячейку влево, что требует в среднем порядка O (n/2) операций копирования, в динамическом – удаление осуществляется за O(1). Существуют два варианта удаления:

а) удаление после текущего.

Пусть tz указывает на некоторый элемент списка, необходимо удалить следующий за ним элемент. Воспользуемся дополнительным указателем p. Установим его на следующее звено, перекинем связь Next минуя удаляемый элемент.**‘с’‘л’‘о’‘н’‘.’**

```
Procedure DelAfter(tz:ref);
Var p:ref;
Begin
   P:=tz^.Next;{p:= след.звено}
   If p<> nil;{если след. есть, то}
   Then begin
            tz^.Next:=p^.Next;
            Dispose(p){перекинем связи и удалим объект}
          end
End;
```

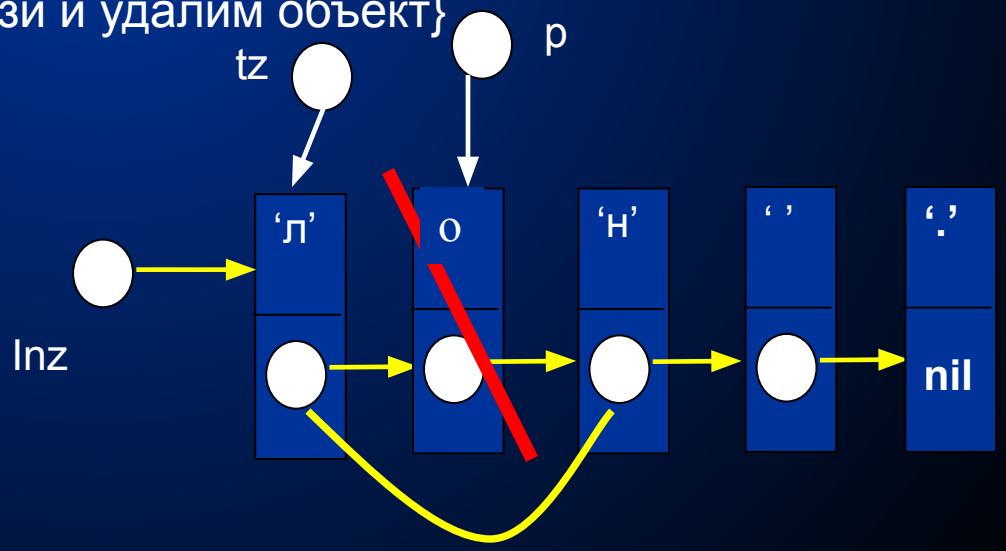
Недостаток метода: нельзя удалить первое звено.



б) удаление текущего.

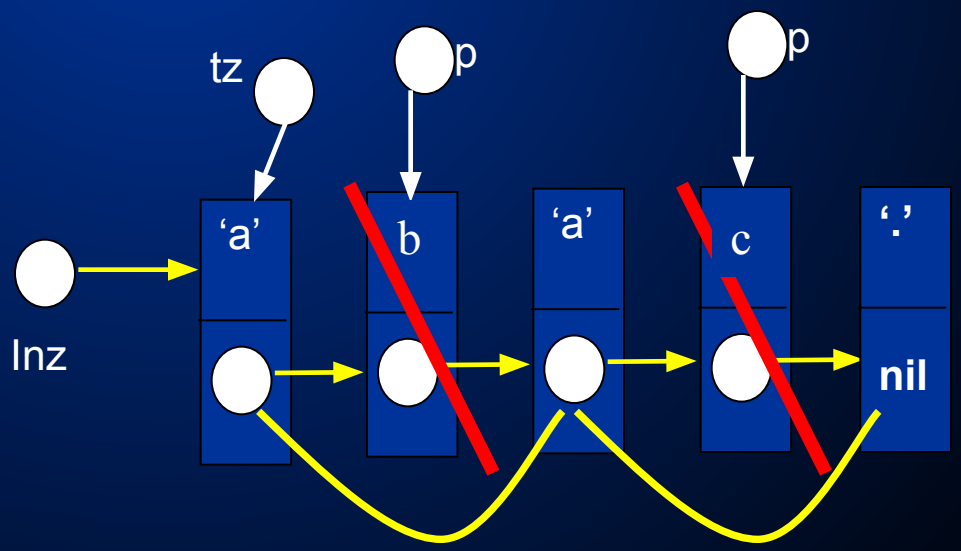
Пусть tz указывает на некоторый элемент списка, который нам необходимо удалить. Опять кажется, что это невозможно, так как мы не имеем доступа к предыдущим элементам и, следовательно, не сможем изменить связи в цепочке. Однако, воспользуемся предыдущей идеей – удалим следующее звено после текущего, предварительно скопировав из него информацию в текущее.

```
Procedure DelCur(tz:ref);
Var p:ref;
Begin
    P:=tz^.Next;{p:= след.звено}
    If p<> nil;{если след. есть, то}
    Then begin
                tz^.Next:=p^.Next;
                tz^.Lit:=p^.lit;
                Dispose(p){перекинем связи и удалим объект}
  end
End;
```



Дан линейный динамический список, удалить в нем все буквы ‘a’.

```
Procedure DelCur(tz:ref);
Var p:ref;
Begin
   P:=tz^.Next;
   If p<> nil;
   Then begin
          tz^.Next:=p^.Next;
          tz^.Lit:=p^.lit;
          Dispose(p)
        end
End;
Procedure DelA(var inz:Ref;
                 a:char);
Vat tz:Ref;
Begin
  tz:=Inz;
  While tz<>nil do
    if tz^.Lit=a
    then DelCur(tz)
    else tz:=tz^.Next
End;
```

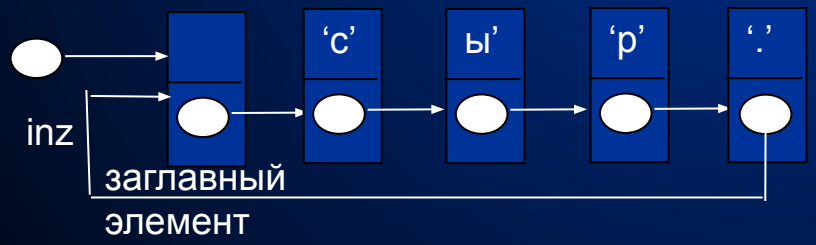


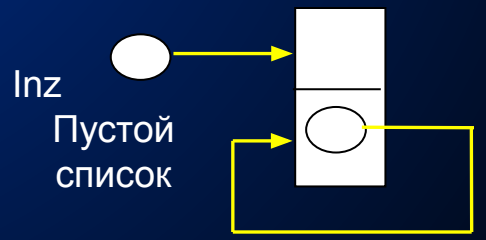
Динамический линейный однонаправленный кольцевой список с заглавным элементом

Основные проблемы классического динамического линейного списка:

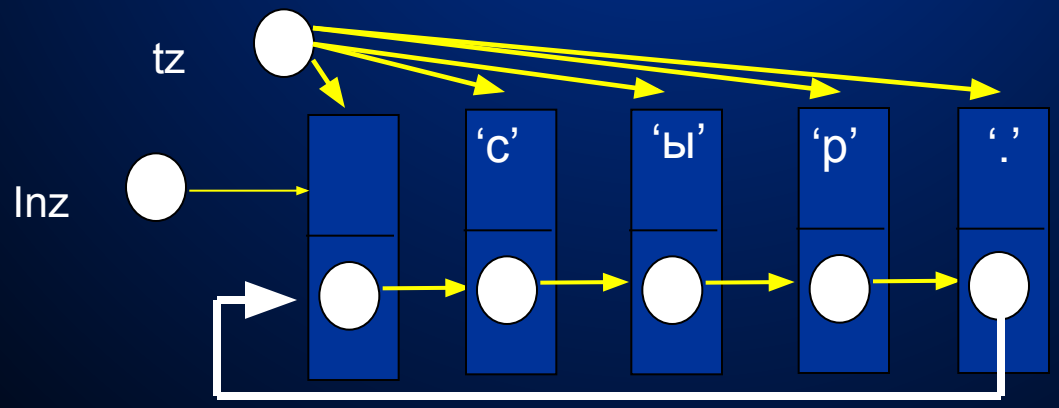
- наличие нескольких частных случаев списка: пустой, один элемент, несколько элементов. Каждый случай требует отдельного рассмотрения;
- при удалении последнего оставшегося звена и получении пустого списка требуется изменение входного указателя inz, а это потребует усложнение процедур удаления элементов;
- трудности с удалением крайних элементов списка;
- гигантские проблемы с пустым списком, вставкой, удалением и так далее.

От большинства этих недостатков избавлен кольцевой список с заглавным элементом. Заглавный элемент не содержит информации, его задача избавиться от частного случая – пустой список, в котором inz=nil. Кольцо – позволяет замкнуть последнюю связь на заглавный элемент, что в принципе позволяет добраться до любого элемента.





```
Procedure CreateRing(var inz:ref);
Var tz:ref; a:char;
Begin
      {создадим заглавное звено}
      New(inz); tz:=Inz;
      repeat
        New(tz^.next);{создадим новое звено}
        Tz:=tz^.Next;{перейдем к следующему звену}
         Read(a); tz^.lit:=a
       Until a='.'
      Tz^.Next:=inz; ;{замкнем конец списка на его начало}
      Readln ;{удалим enter из буфера клавиатуры}
End;
```



Вывод списка

Встаем на начало списка (inz), двигаемся по нему, переходя к следующему элементу (поле Next) и выводим информацию (поле lit).

```
Procedure WriteList(inz:ref);
Var tz:ref;
Begin
     tz:=Inz^.Next;{пропустим заглавный элемент}
    While tz<>Inz Do
      Begin
        write(tz^.Lit);{выведем информацию из текущего звена}
       Tz:=tz^.Next;{перейдем к следующему звену}
    End;
End;
```

Признаком конца списка мы используем не символ ‘.’и пустую ссылку nil, а входной указатель inz.

Поиск элемента в кольцевом списке с заглавным элементом.

Наличие кольца и заглавного элемента позволит нам увеличить скорость поиска в 2 раза. Для этого избавимся от проверки на достижении конца списка, следовательно, процесс остановится только при нахождении ключа. А что делать, если его нет в списке? Разместим искомый в заглавном элементе – он всё равно пустой! Встаем на начало списка (inz), двигаемся по нему, переходя к следующему элементу (поле Next), пока не найдем искомый элемент. Если мы его нашли в заглавном, то в списке искомого элемента не было.

```
function Seek(inz:ref;key:lit):ref;
Var tz:ref;
Begin
    tz:=Inz^.Next; Inz^.Lit:=key;{поместим искомый в заглавный}
    While (tz^.Lit<>key) Do
        Tz:=tz^.Next;{перейдем к следующему звену}
    If tz<>inz then Seek:=tz else Seek:=nil
End;
```

Мы имеем одно сравнение на каждый элемент, поэтому худшая скорость будет O(n), а средняя – O(n/2).

Остальные операции реализуются аналогично.