

Глава 5

Генерация кода

5.1 Генерация внутреннего представления программы

Общая схема распознавателя

5.1.1 Язык внутреннего представления программы

5.1.2 ПОЛИЗ

5.2 Синтаксически управляемый перевод

5.2.1 Генерация внутреннего представления арифметического выражения

5.2.2 Трансляция кода для интерпретации

5.2.3 Генерация кода для оператора READ

5.2.4 Генерация кода для оператора IF

5.2.5 Генерация кода для цикла WHILE

5.2.6 Генерация кода для цикла FOR

5.2.7 Генерация кода для оператора CASE

5.2.8 Генерация кода для цикла с постусловием REPEAT

5.1 Генерация внутреннего представления программы

- Результатом работы синтаксического анализатора должно быть некоторое представление программы, которое отражает ее синтаксическую структуру.
- Программа в таком представлении дальше может либо интерпретироваться либо транслироваться в объектный код.

5.1.1 Язык внутреннего представления программы

- Свойства:
 - Он позволяет фиксировать синтаксическую структуру программы.
 - Текст на нем можно автоматически генерировать на этапе синтаксического разбора.
 - Его конструкции должны достаточно просто транслироваться в объектный код либо достаточно эффективно интерпретироваться.

5.1.1 Язык внутреннего представления программы

- Некоторые общепринятые способы внутреннего представления программы:
 - Постфиксная запись;
 - Префиксная запись;
 - Многоадресный код с неявно именуемыми результатами (триады);
 - Многоадресный код с явно именуемыми результатами (тетрады);
 - Связные списочные структуры, представляющие деревья операций.

Пример

$\langle \text{stmt} \rangle ::= \langle \text{id} \rangle := \langle \text{expr} \rangle$

$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle$

$\langle \text{term} \rangle ::= \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{factor} \rangle$

$\langle \text{factor} \rangle ::= (\langle \text{expr} \rangle) \mid \langle \text{id} \rangle$

A:=B*C+D

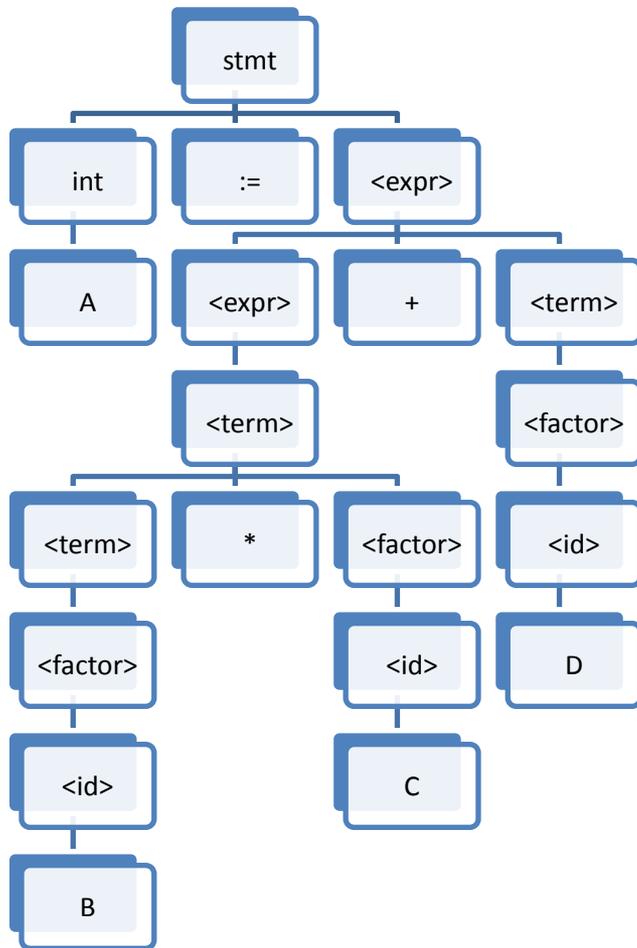
ABC*D+:=

Польская инверсная запись
*, B, C
+, (1), D
:=, (2), A
Триады

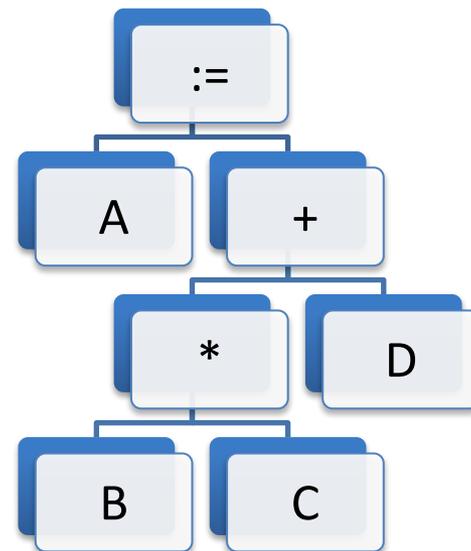
*, B, C, t1
+, t1, D, t2
:=, t2, null, A
Тетрады

Пример

Дерево синтаксического разбора



Дерево операций



5.1.2 ПОЛИЗ

- В полизе операнды выполняются слева направо в порядке их следования (в инфиксной записи), знаки операций размещают таким образом, что знаку операции непосредственно предшествуют операнды, скобки отсутствуют.

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

ПОЛИЗ: $abc-*d/ef+g*-$

- Формальное определение постфиксной записи:
 - Если E – единственный операнд, то полизом такого выражения будет этот операнд
 - Если есть выражение вида $E_1 \Theta E_2$, где Θ – бинарная операция, то $E_1' E_2' \Theta$, где E_1', E_2' – полизы E_1 и E_2 .
 - Если ΘE , где Θ – знак унарной операции, то $E' \Theta$, где E' – полиз E .
 - Полизом выражения (E) будет E' , где E' – полиз E .

5.1.2 ПОЛИЗ

Алгоритм интерпретации Полиза.

Используем стек, выражения читаем слева направо. Если очередным элементом полиза является операнд, то заталкиваем в стек. Если операция, то из стека выталкиваем необходимое количество операндов, проводим вычисления и результат заталкиваем в стек.

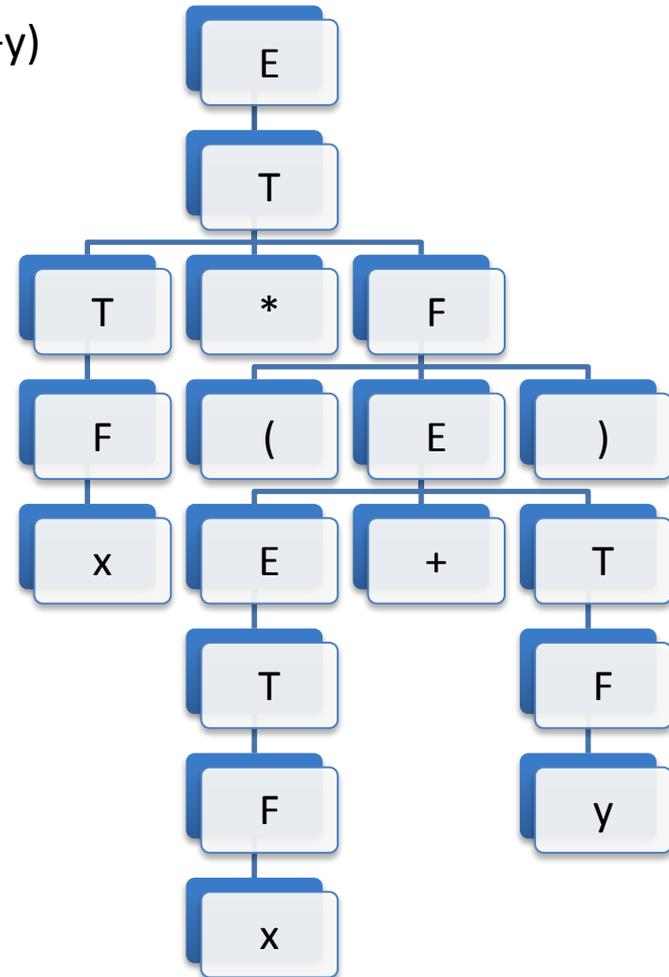
5.2 Синтаксически управляемый перевод

- На практике синтаксический, семантический анализ и генерация внутреннего представления программы осуществляется одновременно. Существует несколько способов, один из них – синтаксически управляемый перевод.
- В основе СУ – перевода лежит способ сопоставления правилам грамматики соответствующих процедур генерации (семантические подпрограммы).

5.2.1 Генерация внутреннего представления арифметического выражения

1	$E \rightarrow E+T$	printf (“+”)	
2	$E \rightarrow T$		
3	$T \rightarrow T * F$	printf (“*”)	
4	$T \rightarrow F$		
5	$F \rightarrow (E)$		
6	$F \rightarrow x$	printf (“x”)	
7	$F \rightarrow y$	printf (“y”)	

$x*(x+y)$



5.2.1 Генерация внутреннего представления арифметического выражения

- Левосторонний вывод

$$E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow x * F \Rightarrow x * (E) \Rightarrow x * (E + T) \Rightarrow x * (T + T) \Rightarrow x * (F + T) \Rightarrow x * (x + T) \Rightarrow x * (x + F) \Rightarrow x * (x + y)$$

2 3 4 6 5 1 2 4 6 4 7

* x + x y префиксная запись

- Правосторонний вывод

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * (E) \Rightarrow T * (E + T) \Rightarrow T * (E + F) \Rightarrow T * (E + y) \Rightarrow T * (T + y) \Rightarrow T * (F + y) \Rightarrow T * (x + y) \Rightarrow F * (x + y) \Rightarrow x * (x + y)$$

6 4 6 4 2 7 4 1 5 3 2

x x y + * постфиксная запись

5.2.2 Трансляция кода для интерпретации

1	$E \rightarrow E+T$	printf (“+”)	[call xADD] add esp,8 push eax
2	$E \rightarrow T$		
3	$T \rightarrow T * F$	printf (“*”)	[call xMUL] add esp,8 push eax
4	$T \rightarrow F$		
5	$F \rightarrow (E)$		
6	$F \rightarrow x$	printf (“x”)	push x
7	$F \rightarrow y$	printf (“y”)	push y

5.2.2 Трансляция кода для интерпретации

```
xADD proc near
    pop bp; адрес возврата
    pop ax; первый операнд
    pop bx; второй операнд
    ADD ax,bx; сложение
    Push ax; результат в стек
    Push bp; адрес возврата в
стек
    ret ; возврат
xADD endp
```

Для выражения $x(x+y)$*

```
push x
push x
push y
call xADD
add esp, 8
push eax
call xMULL
add esp, 8
push eax
```

5.2.2 Трансляция кода для интерпретации

- Про операцию присвоения

$I := E$

В полIZE $IE' :=$,

где I – адрес, E' – полIZE E

Правило	Процедура генерации
1. $\langle \text{assign} \rangle ::= \langle \text{id} \rangle := \langle \text{expr} \rangle$	[call xASSIGN]
2. $\langle \text{id} \rangle ::= \text{id}$	[push offset ID]

proc xASSIGN

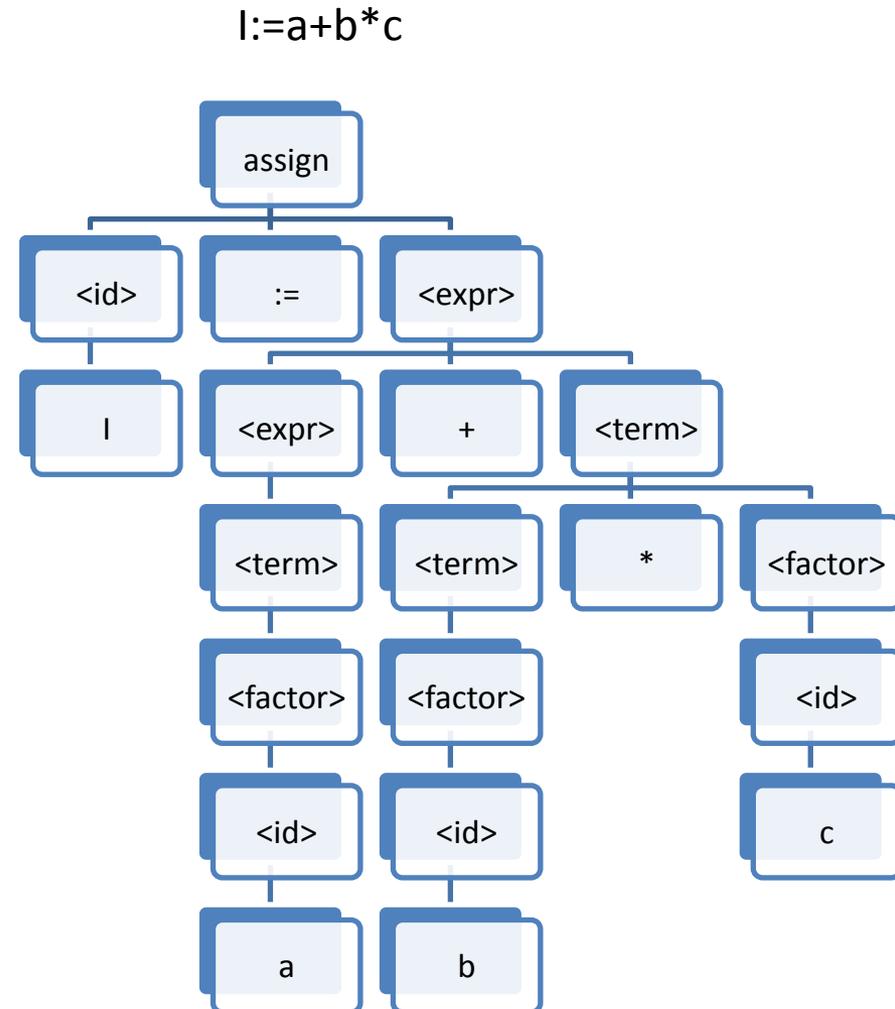
pop bp

pop ax

mov [bx], ax

push bp

ret



Пример написания семантических процедур

Дана грамматика для описания дробных чисел с точкой. Обеспечить перевод числа таким образом, чтобы целая часть стала дробной, а дробная – целой.

1 <десят. с фиксир. точкой> := <целое> <.> <целое>

2 <.> := .

3 <целое> := <целое> <цифра>

4 <целое> := <цифра>

5 <цифра> := 0 | 1 | ... | 9

0 { int f=0; }

2 { f=1; }

5 { if (f) printf(“%c”, c);
 else q.enqueue(c); // добавить в очередь}

1 { printf(“.”); while (!q.queue()) printf(“%c”, q.dequeue);}

5.2.3 Генерация кода для оператора READ

```
READ (A,S);
```

```
push offset A
push offset S
push 2
call xREAD
add sp, 2*(2+1)
```

```
proc near xREAD
    mov ex, [sp+2]
    lea bx, [sp+2+ex*2]
```

```
@L1:
    call xREAD_AX
    mov [bx], ax
    sub bx, 2
    loop @l1
ret
```

1. `<read> := READ (<id_list>);`
2. `<id_list> := <id_list>, <id>`
3. `<id_list> := <id>`
4. `<id> := _ID_`

```
0 { int arg_count; }
4 { arg_count++;
  [ push offset $ in ] }
1 { push offset $ arg_count
  [call xREAD]
  [add sp, 2*(arg_count+1)]}
```

5.2.4 Генерация кода для безусловного перехода

- **goto L**
[jmp L]
[L:]
- 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.

5.2.5 Генерация кода для оператора IF

- 1.<оператор if> ::= IF <условие if> THEN <блок then>
- 2.<оператор if> ::= IF <условие if> THEN <блок then> ELSE <блок else>
- 3.<условие if> ::= <expr L>
- 4.<блок then> ::= <блок>

1. If B then S

Введем вспомогательную операцию - условный переход "по лжи" с семантикой

if (not B) then goto L

Это двухместная операция с операндами B и L. Обозначим ее !F

ПОЛИЗ: B' p !F

где p - номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L.

2. if B then S1 else S2

if (not B) then goto L2; S1; goto L3; L2: S2; L3: ...

ПОЛИЗ: B' p2 !F S1' p3 ! S2' ...

5.2.5 Генерация кода для оператора IF

if (условие)	OR ax,ax jnz метка_XXX jmp метка_YYY метка_XXX:	Генерируем код с уникальными именами меток, имена этих меток ложим в стек
... then...	jmp метка_ZZZ метка_YYY:	Здесь мы достаём из стека имя метки, вставляем эту метку в ассемблеровский листинг, затем ложим в стек новую уникальную метку
... else...;	метка_ZZZ:	Достаём из стека ярлык с меткой, которую вставляем в листинг

```
if [что-то] then [это] else [то]
```

```
    [код для <что-то>]
```

```
POP AX
```

```
OR AX, AX
```

```
jnz adr001
```

```
jmp adr002
```

```
adr001:
```

```
    [код для это]
```

```
jmp adr003
```

```
adr002:
```

```
    [код для то]
```

```
adr003:
```

5.2.5 Генерация кода для оператора IF

```
if [что-то1] then
    if [что-то2] then
        if [что-то3] then [это3] else [то3]
    else [то2]
```

```
    [код для <что-то1>]
POP AX
OR AX, AX
jnz adr001
jmp adr002
adr001:
    [код для <что-то2>]
POP AX
OR AX, AX
jnz adr003
jmp adr004
adr003:
    [код для <что-то3>]
POP AX
OR AX, AX
```

```
jnz adr005
jmp adr006
adr005:
    [код для это-3]
jmp adr007
adr006:
    [код для то-3]
adr007:
    jmp adr008
adr004:
    [код для то-2]
adr008:
    jmp adr009
adr002:
adr009:
```

5.2.6 Генерация кода для цикла

WHILE

Семантика оператора цикла while B do S может быть описана так:

L0: if (not B) then goto L1; S; goto L0; L1:

ПОЛИЗ : B' p1 !F S' p0 ! ... ,

где p_i - номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L_i , $i = 0, 1$

5.2.6 Генерация кода для цикла

WHILE

```
While <что-то> do
begin
    <операторы>
end
```

1. <оператор w>::=<while> <услов. w> DO <body>
2. <while>::=WHILE
3. <услов. w>::=<exprL>
4. <body>::=<stmt> | BEGIN <stmt_list> END

ПОЛИЗ: 2
3 4 1

```
2. @L0:
   [код для что-то]
3. POP AX
   OR AX, AX
   jnz @L1
   jmp @L2
   @L1:
4. [код для операторы]
1. jmp @L0
   @L2:
```

```
2. генерируем уникальную метку L0
   s.push(L0)
   [$L0:]
3. генерируем уникальную метку L1,
   L2
   s.push(L1)
   s.push(L2)
   [POP AX]
   [OR AX, AX]
   [jnz $L1]
   [jmp $L2]
   [$L1: nop]
1. L2=s.pop ( )
   L0=s.pop ( )
   [jmp $L0]
   [$L2: nop]
```

5.2.6 Генерация кода для цикла

WHILE

```
While <что-то> do
  While <еще что-то> do
    begin
      <операторы>
    end
```

ПОЛИЗ: 2 3 2 3 4 1 4

1

```
@adr0032:
  [код для что-то]
  POP AX
  OR AX, AX
  jnz @adr0033
  jmp @adr0034
@adr0033:
@adr0035:
  [код для еще что-то]
  POP AX
  OR AX, AX
  jnz @adr0036
  jmp @adr0037
@adr0036:
  [код для операторы]
  jmp @adr0035
@adr0037:
  jmp @adr0032
@adr0034:
```

```
@adr0034
@adr0033
@adr0032
```

```
@adr0034
@adr0032
```

```
@adr0037
@adr0036
@adr0035
@adr0034
@adr0032
```

```
@adr0037
@adr0035
@adr0034
@adr0032
```

```
@adr0034
@adr0032
```

5.2.7 Генерация кода для цикла

FOR

FOR i:=1 TO N do S

B1 => I <= N

B2 => I < N

I => i++

if (not B1) then goto L2; goto L1; L0: I; L1: S

if (not B2) then goto L2; goto L0;

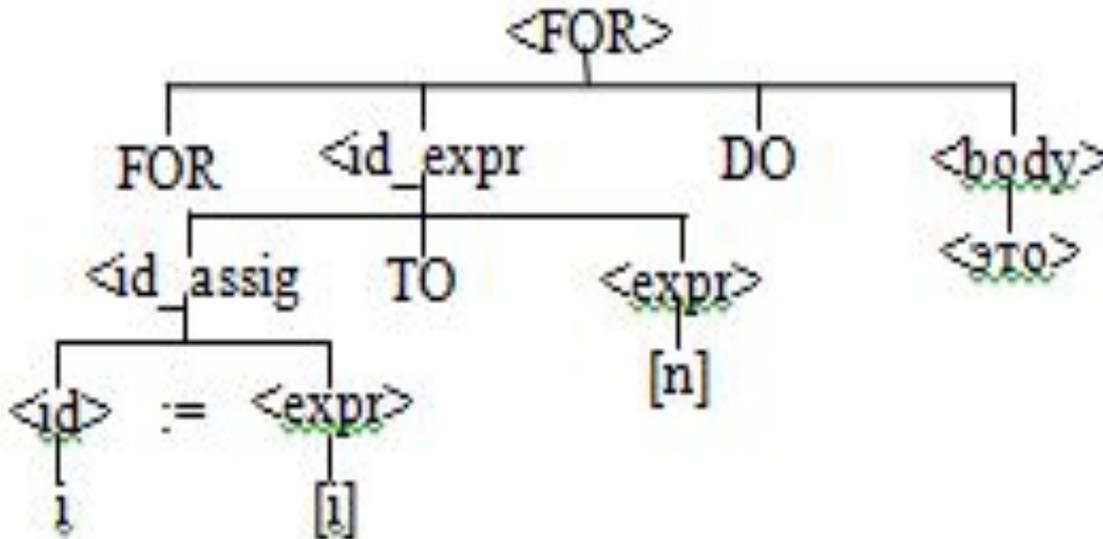
ПОЛИЗ: B1' p2 !F p1 ! I' S' B2' p2 !F p0 !

5.2.7 Генерация кода для цикла

FOR

FOR i:=<что-то> TO <еще что-то> do

<это>



FOR i=1 to n do <body>

L1:

[проверка условия]

jmp L2

[выполнение это]

inc counter

jmp L1

L2:

1. <for>::=FOR<idx_expr> DO<body>
2. <idx_expr>::=<id_assign> TO <expr>
3. <body>::=<stmt>|BEGIN <stmt_list> END
4. <id_assign>::=<id>:=<expr>

ПОЛИЗ: 4 2

3 1

5.2.7 Генерация кода для цикла

FOR

```
4. [pop id]
   v.push(id)

2. [pop DI]
   генерировать L1, L2, L3
[L1:]
   _i:=v.pop()
   [cmp _i, DI]
   [jng L2]
   [jmp L3]
[L2:]
   [push DI]
   s.push(L3);
   s.push(L1)
1. [pop DI]
   i=v.pop
   [inc _i]
   L1=s.pop()
   [jmp L1]
   L3=s.pop()
[L3:]
```

```
[сформулировался код для <expr>]
   pop _i
   [код для expr2]
   pop DI
adr001:
   cmp _i, DI
   jng adr002
   jmp adr003
adr002:
   push DI
   [код <body>]
   pop DI
   inc _i
   jmp adr001
adr003: nop
```

5.2.7 / енерация кода для цикла

FOR

```
FOR i:=<что-то> TO <еще что-то> DO
  FOR j:=<что-то2> TO <еще что-то2>
  DO
```

<это>

ПОЛИЗ: 4 2 4 2 3 1

[код для что-то1]

pop _i

[код для ещё что-то1]

pop DI

adr001:

cmp _i, DI

jng adr002

jmp adr003

adr002:

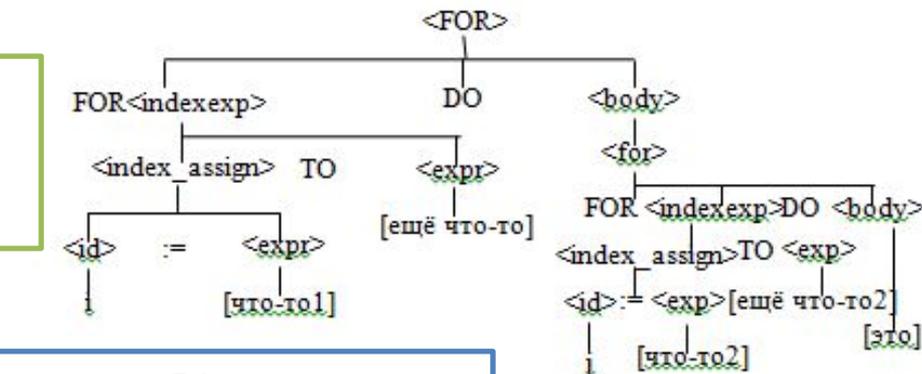
push DI

[код для что-то2]

pop _j

[код для ещё что-то2]

pop DI



cmp _j, DI

jng adr005

jmp adr006

adr005:

push DI

[код для это]

pop DI

inc _j

jmp adr004

adr006:

pop DI

inc _i

jmp adr001

adr003:

nop

5.2.8 Генерация кода для оператора

CASE

1. `<case_stmt> ::= CASE <tag> OF <variant_list> END;`
2. `<tag> ::= <expr>`
3. `<variant_list> ::= <variant>; <variant_list> | <variant>`
4. `<variant> ::= <case_const> : <body>`
5. `<body> ::= <stmt> | BEGIN <stmt_list> END`

CASE N OF

```
1: S1;  
2: S2  
END;
```

```
2. генерируем уникальную метку  
LCEND  s.push(LCEND)  
      [POP AX]  
5.  cmp ax, $int  
генерируем уникальные метки L1, L2  
      [je @L1]  
      [jmp @L2]  
      [$L1: ]  
      s.push (L2)  
4.  L2=s.pop ( )  
      LCEND =s.pop ( )  
      [jmp $LCEND]  
      [$L2: nop]  
1.  LCEND =s.pop ( )  
      [$LCEND : nop]
```

ПОЛИЗ: 2 5 4 5 4 3

```
1  
2.  [КОД ДЛЯ N]  
      POP AX  
5.  cmp ax, 1  
      je @L1  
      jmp @L2  
      @L1:  
      ; S1  
4.  jmp LCEND  
      @L2:  
      cmp ax, 2  
      ...  
      LCEND :
```

5.2.9 Генерация кода для цикла с постусловием REPEAT

REPEAT Si UNTIL B

L0: S; if (not B) then goto L0;; ...

ПОЛИЗ: S' B' p0 !F ...

1. <оператор R>::=<REPEAT> <body> UNTIL <усл. R>

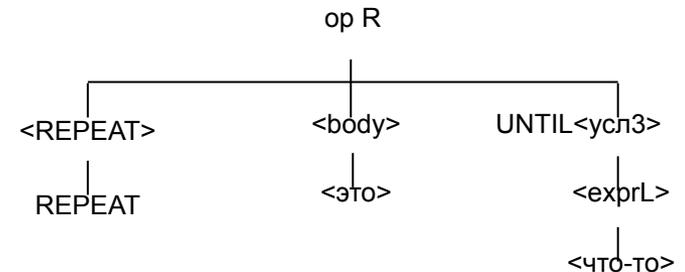
2. <REPEAT>::=REPEAT

3. <усл. R>::=<exprL>

4. <body> ::= <stmt> | REPEAT <stmt list> END

REPEAT <ЧТО-ТО> UNTIL

<ЭТО>



ПОЛИЗ: 2 4

3 1

2. генерировать L1:

s.push (L1)

[L1:]

3. [POP AX]

[OR AX, AX]

генерируем L2

[jnz L2]

L1:=s.pop ();

[jmp L1]

[L2:]

[L1:]

[код для <это>]

[вычисление условий]

pop AX

OR AX, AX

jnz L2

jmp L1

L2:

5.2.10 / енерация кода для раздела объявления

ПЕРЕМЕННЫХ

1. <program> ::= <prog> <prog_name> <var> <dec_list> <begin> <stmt_list> END.
2. <prog>:= PROGRAM
3. <prog_name> ::= <id>
4. <var> ::= VAR
5. <dec_list> ::= <dec> | <dec_list> ; <dec>
6. <dec> ::= <id_list> : <type>
7. <type> ::= INTEGER
8. <id_list> ::= <id> | <id_list> , <id>
9. <begin> ::= BEGIN
10. <end> ::= END.
11. <stmt_list> ::= <stmt> | <stmt_list> ; <stmt>
12. <stmt> ::= <assign> | <read> | <write> | <for> | <case> | <while> | <repeat> | <if>

ПОЛИЗ: 2 3 4 8 7 6 5 ... 1

8. 7. [_ID DW 0]

5. [DSEG ENDS]

1. [CSEG ENDS]
[end start]

3. [START: JMP BEGIN]

4. [DSEG SEGMENT]

9. [BEGIN:]

2. [p386]
[model tiny]
[CSEG SEGMENT]
[ASSUME CS:CSEG]
[ORG 100H]

Пример. Генерация кода

```
PROGRAM MYPROG;  
VAR  
INT1, INT2 : INTEGER;  
BEGIN  
  For INT1:=1 to 10 do  
    begin  
      INT2:=(INT1+1)*(INT1+2);  
    end;  
    write(INT2);  
  END.
```

```
p386  
model tiny  
CSEG SEGMENT  
org 100h  
start: jmp begin  
      Dseg SEGMENT  
      INT1 dw 0  
      INT2 dw 0  
      DSEG ENDS  
begin:  
  push offset INT1  
  push 1  
  call xassign  
@cmp_0:  
  push 10  
  pop ax  
  cmp ax, [INT1]  
  jge @for_0  
  jmp @end_0
```

```
@for_0:  
  push offset INT2  
  push [INT1]  
  push 1  
  call xadd  
  push ax  
  push [INT1]  
  push 2  
  call xadd  
  push ax  
  call xmul  
  call xassign  
  inc [INT1]  
  jmp @cmp_0  
@end_0:  
  push [INT2]  
  call write_word  
  cseg endseg  
end start
```