Автоматы и формальные языки

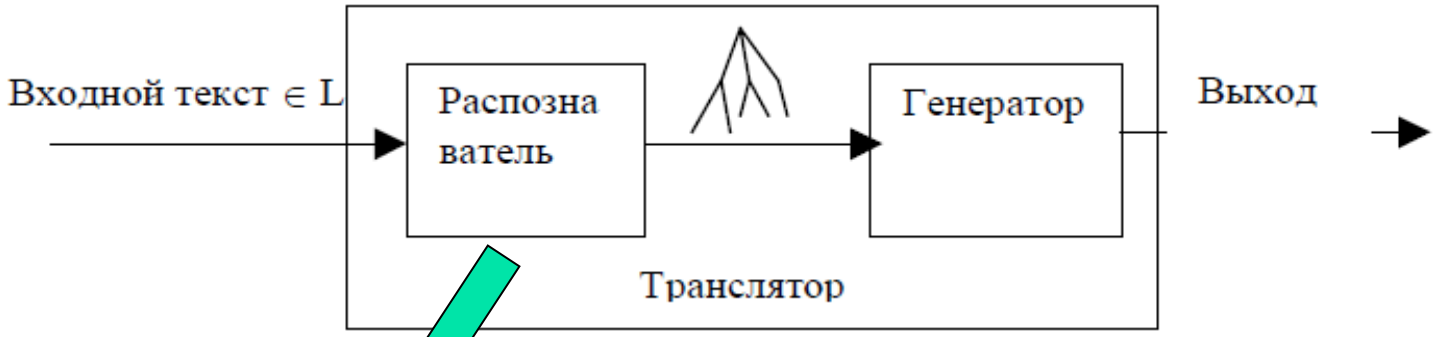
Карпов Юрий Глебович
профессор, д.т.н., зав.кафедрой
“Распределенные вычисления и компьютерные сети”
Санкт-Петербургского Политехнического университета

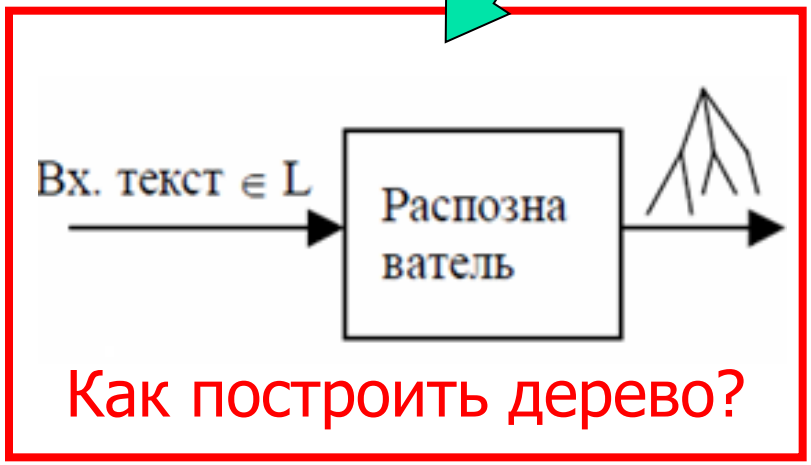
karpov@dcn.infos.ru

Структура курса

- Конечные автоматы-распознаватели – 4 л
- Порождающие грамматики Хомского – 3 л
- Атрибутные трансляции и двусмысленные КС-грамматики – 2 л
- Распознаватели КС-языков и трансляция – 6 л
  - Лекция 10. s-грамматики, LL(k)-грамматики, грамматики рекурсивного спуска
  - Лекция 11. Построение транслятора языка Милан методом рекурсивного спуска
  - Лекция 12. Грамматики предшествования, LR(k)-грамматики
  - Лекция 13. SLR(k) и LALR(k)-грамматики.
  - Лекция 14. Компиляторы компиляторов. Yacc и Bison.
  - Лекция 15. Грамматики Кока-Янгера-Касами и Эрли
- Дополнительные лекции - 2 л

Задача синтаксического анализа

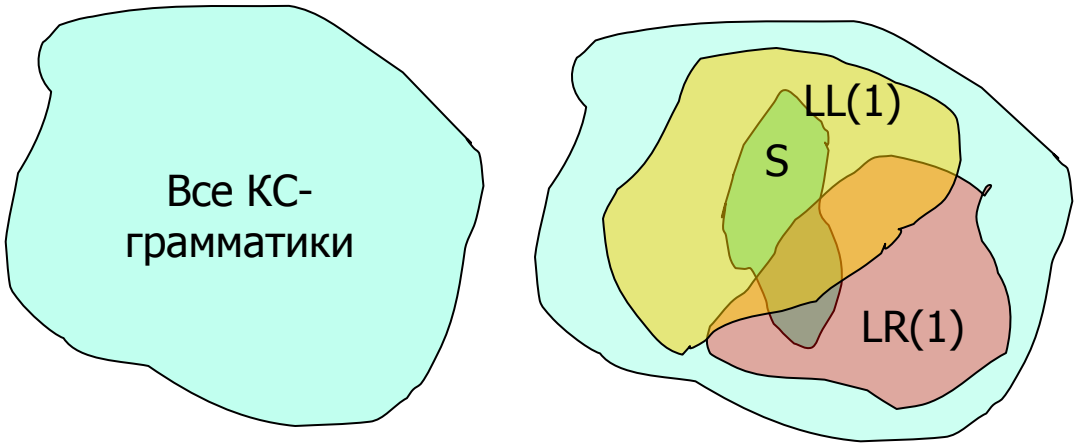




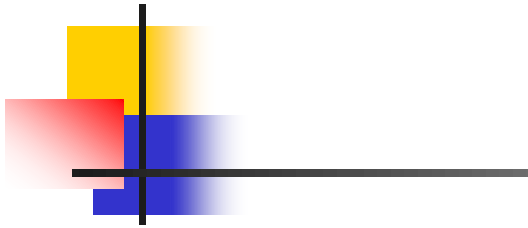
Как построить дерево вывода любой цепочки языка, порождаемого заданной грамматикой??

Синтаксический анализ - одна из наиболее глубоко проработанных и понятных ветвей Computer Science

Существуют "универсальные"алгоритмы СА, работающие для любой КС-грамматики, но они неэффективны



Подклассы КС-грамматик, для которых разработаны ЭФФЕКТИВНЫЕ алгоритмы синтаксического анализа:
s-грамматики
LL(k)-грамматики,
Грамматики рекурсивного спуска
Грамматики предшествования
LR(k)-грамматики,
LALR(k)-грамматики,
...

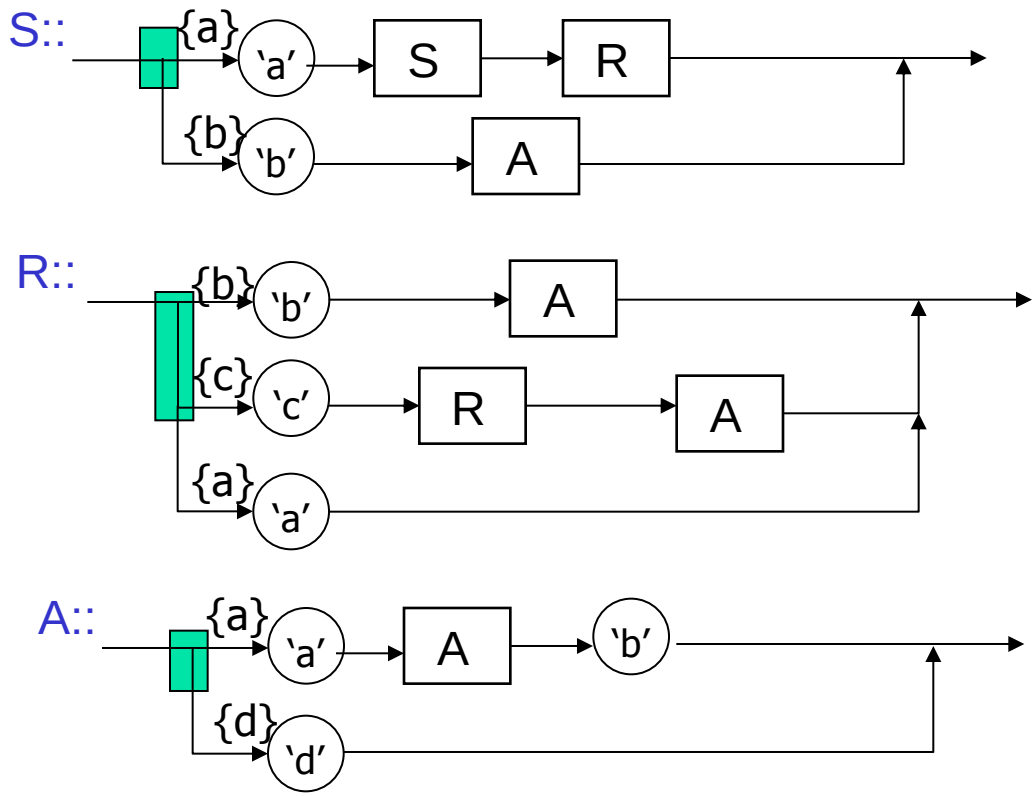


Грамматики рекурсивного спуска

Грамматики рекурсивного спуска: Определение

- Грамматика рекурсивного спуска представляется синтаксическими диаграммами для каждого нетерминала

1. S → aSR
2. S → bA
3. R → bA
4. R → cRA
5. R → a
6. A → aAb
7. A → d



Синтаксические диаграммы будут представлять АЛГОРИТМ РАСПОЗНАВАНИЯ,
если в любом разветвлении каждой синтаксической диаграммы выбор пути может быть сделан на основе анализа одного очередного символа входной цепочки

Синтаксические диаграммы – механизм порождения цепочек языка

Идея алгоритма рекурсивного спуска

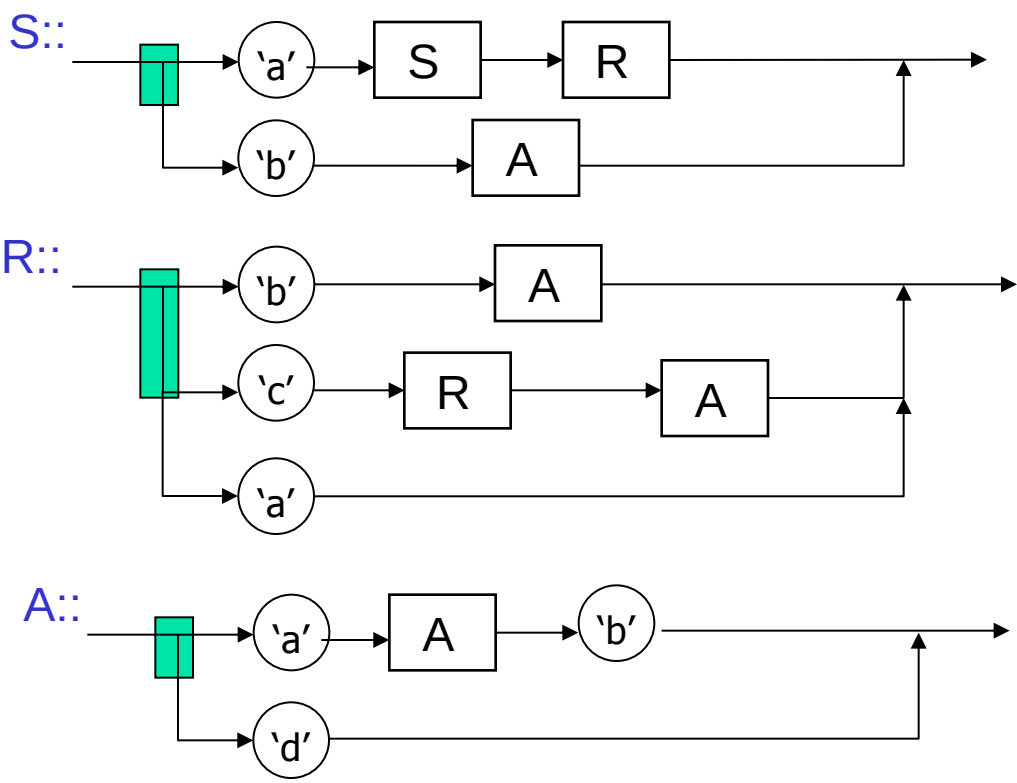
```
int Ук:=0; // указ-ль на очередной символ

S ( ) {
      if( s[Ук] == ‘a’) then { Ук++; S( ); R( ) }
      else
       { if( s[Ук] == ‘b’) then { Ук++; A( ) }
       else { Error1( ) }
        }
 }

R ( ) { ...  ...  ... }

A ( ) {
 if( s[Ук] == ‘a’) then { Ук++;  A( );
      if( s[Ук] == ‘b’) then { Ук++ }
           else { Error2( ) }
      }
 else { if( s[Ук] == ‘d’) then { Ук++ }
         else { Error3( ) };
       }
}
```

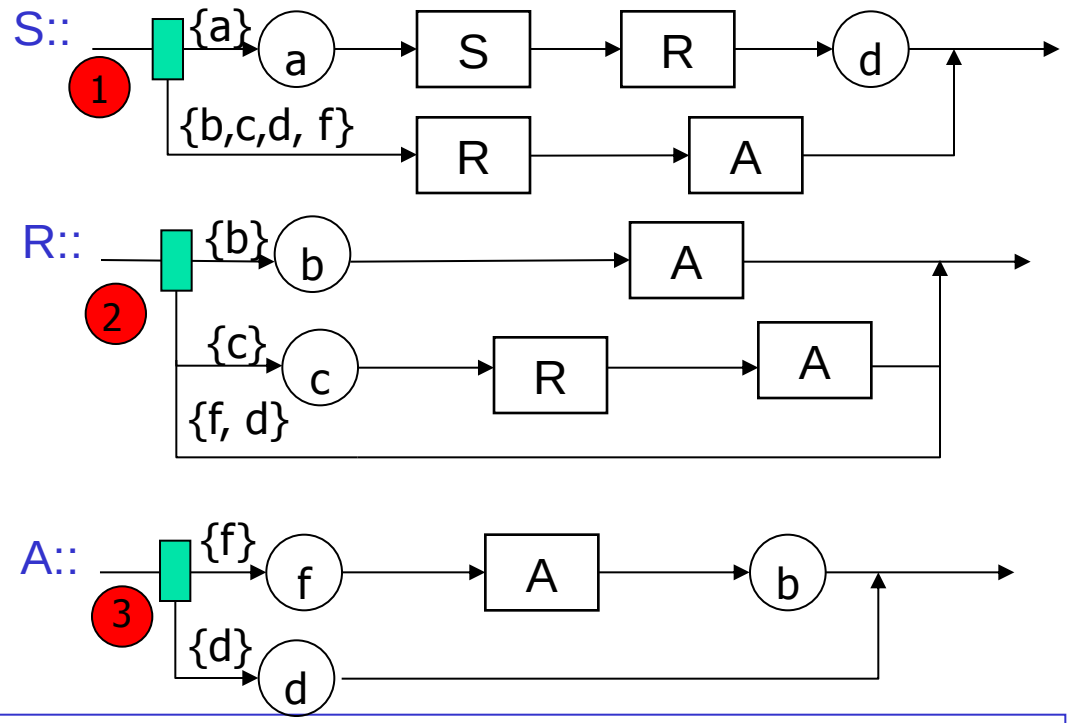
При входе в процедуру N Ук установлен на первый символ подцепочки, выводящейся из нетерминала N



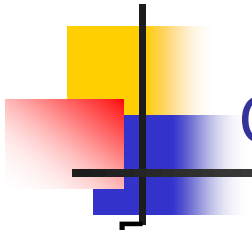
- Каждому нетерминалу сопоставляется процедура. Начальному нетерминалу – процедура main
- Каждая процедура распознает максимальную подстроку своего языка

Грамматика рекурсивного спуска (НО не s-грамматика)

1. S → aSRd
2. S → RA
3. R → bA
4. R → cRA
5. R → ε
6. A → fAb
7. A → d



- Выбор 1 в S: второй из возможных путей определяется терминалами {b, c, f, d}, т.е. теми, с которых могут начаться цепочки, выводимые из R, и теми, которые могут встретиться после R, поскольку есть правило R → ε. Выбор однозначен
- Выбор 2 в R: два пути – однозначно по терминалу b или c. Третий из возможных путей (правило 5) определяется терминалами Follow(R), т.е. теми, которые могут встретиться после R. Это множество {f, d}. Выбор пути однозначен: {f, d} не пересекается с {b, c}
- Выбор 3 в A: определяется очередным терминалом f или d. Выбор однозначен



Определение

- Грамматикой рекурсивного спуска называется LL(1)-грамматика, которая представляется такими синтаксическими диаграммами, что в любом разветвлении каждой синтаксической диаграммы выбор альтернативного пути может быть сделан на основе анализа одного очередного терминального символа

Рекурсивный спуск, по мнению многих авторов, является единственным методом, полностью пригодным для ручного написания компилятора для языков средней сложности (типа mini-C)

Пример. Язык перечисления номеров страниц

Pages: 23, 13-54, 128-97

цепочка

Структура:

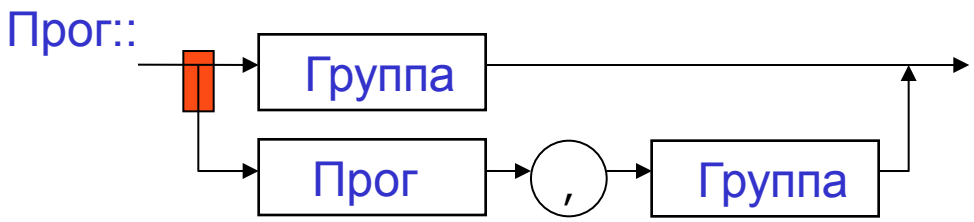
Цепочка - это:
Группы, разделенные запятыми

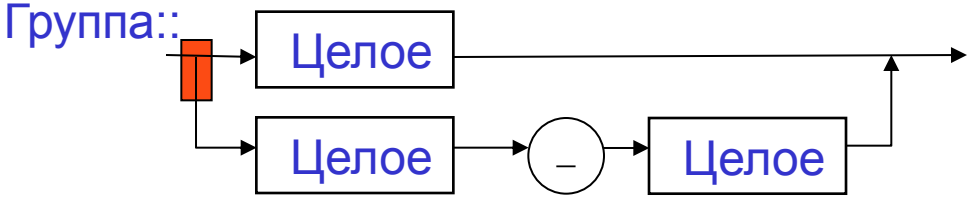
Группа - это:
Целое, либо Целое тире Целое

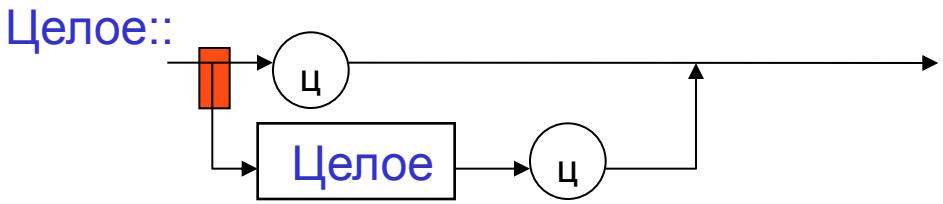
Целое – это:
непустая последовательность цифр

Грамматика:

Прог → Группа | Прог , Группа
Группа → Целое | Целое - Целое
Целое → ц | Целое ц







Как однозначно выбрать путь по синтаксической диаграмме??

Это грамматика с прямой левой рекурсией. Ее синтаксические диаграммы не могут служить схемой алгоритма распознавания

Эквивалентное преобразование грамматики языка перечисления номеров страниц

Pages: 23, 13-54, 128-97

Структура:

Цепочка - это:
Группы, разделенные запятыми

Группа - это:
Целое, либо Целое тире Целое

Целое – это:
непустая последовательность цифр

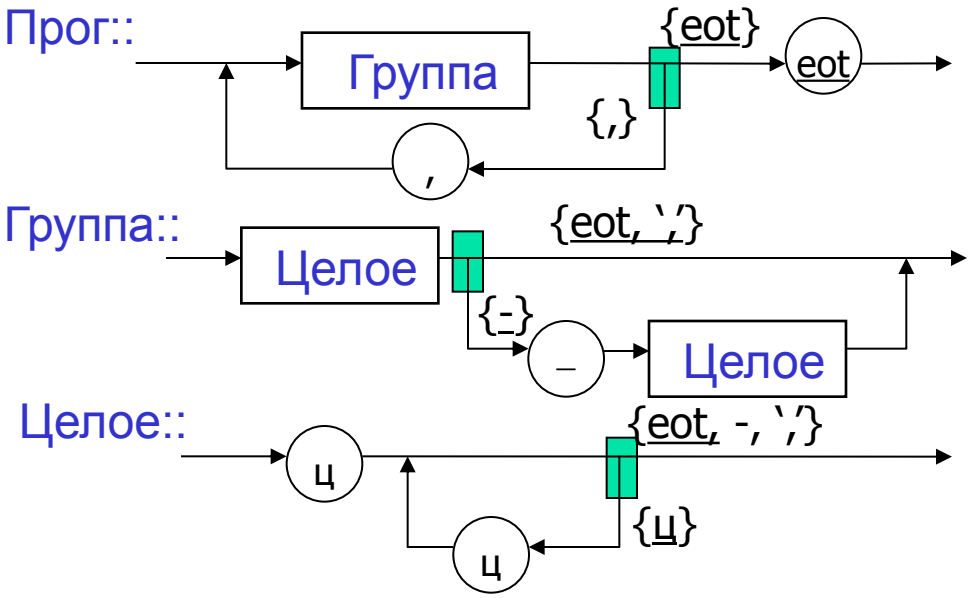
Грамматика:

Прог → Группа | Прог , Группа
Группа → Целое | Целое - Целое
Целое → ц | Целое ц

Убираем левую рекурсию и факторизуем

Грамматика (в БНФ):

Прог → Группа { , Группа } eot // 0 или k раз
Группа → Целое [- Целое] // 0 или 1 раз
Целое → ц { ц }



Эти синтаксические диаграммы однозначны (они могут служить схемой алгоритма распознавания)

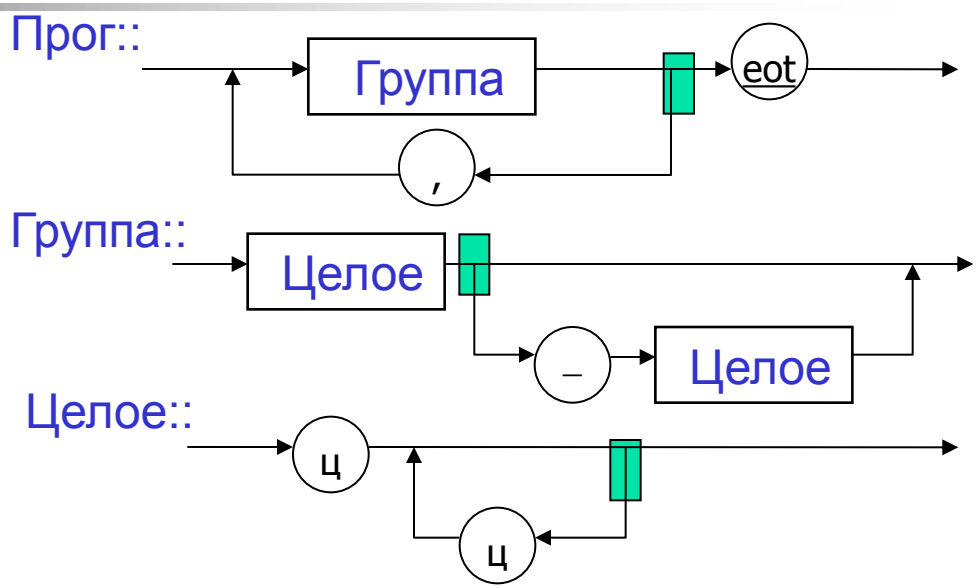
Синтаксические диаграммы языка перечисления номеров страниц

Pages: 23, 13-54, 128-97

Цепочка - это:
Группы, разделенные запятыми

Группа - это:
Целое, либо Целое тире Целое

Целое – это:
непустая последовательность цифр

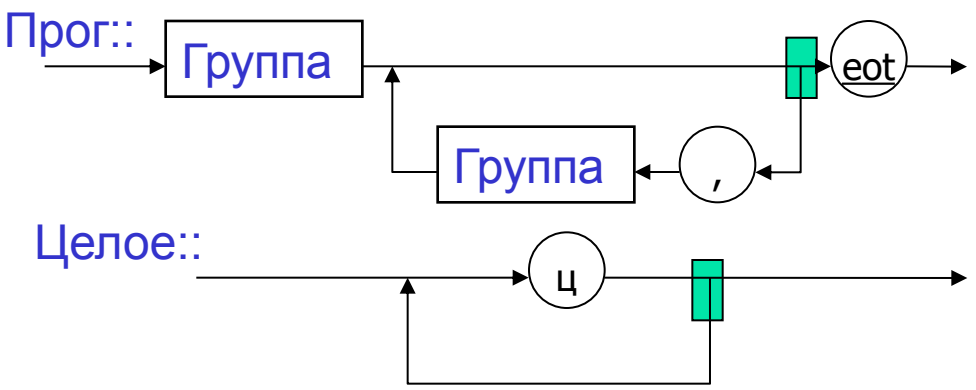


Убираем левую рекурсию и факторизуем

Грамматика:

Прог → Группа { , Группа } eot
Группа → Целое [- Целое]
Целое → ц { ц }

Можно немного по-другому



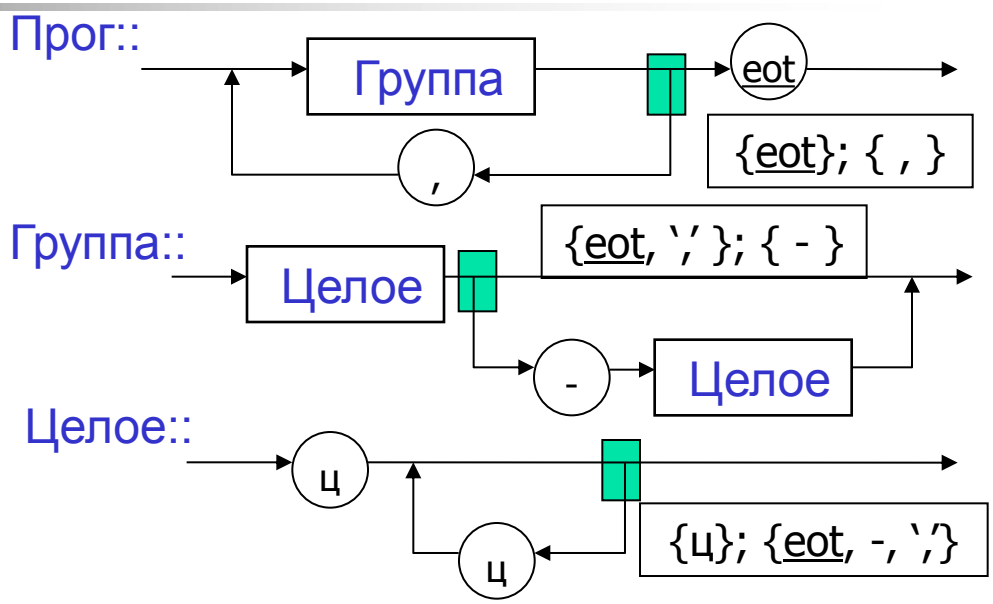
Распознаватель языка перечисления номеров страниц

```
int Ук!=0; // указ-ль на очередной символ

main( ) {
m: Группа( );
    if( s[Ук] == 'eot') then { Ук++; return };
      else if( s[Ук] == ',') then { Ук++; goto m };
      else  { Error1( )}
};

Группа( ) {
 Целое( );
  if( s[Ук] == '-') then { Ук++; Целое() };
     else { return }
};

Целое( ) {
       if( s[Ук] == 'ц') then { Ук++ ; }
       else { Error2( ) };
  m: if( s[Ук] == 'ц') then { Ук++; goto m};
      else { return  }
};
```



Для каждого нетерминала – своя распознающая процедура

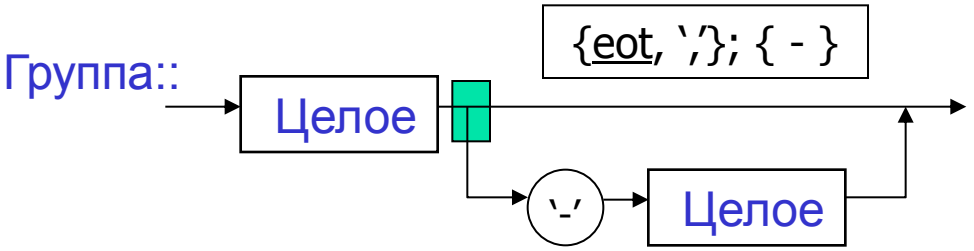
Эта грамматика рекурсивного спуска: в каждом разветвлении синтаксической диаграммы однозначно определяем путь по очередному терминальному символу

Добавление семантических операций в синтаксическую диаграмму позволяет построить транслятор

Правила работы с указателем очередного символа входа

Процедура, распознающая цепочку, выводимую из нетерминала Группа

```
Группа( ) {
   Целое( );
  if( s[Ук] == '-') then { Ук++; Целое( ) };
   else { return }
}
```

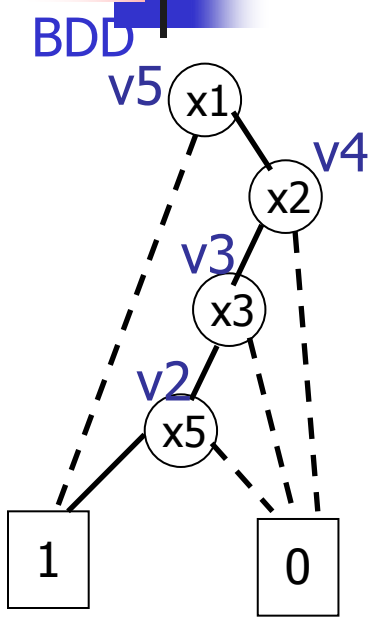


1. Вначале, при входе в процедуру main (нач. нетерминал), указатель устанавливается на первый символ входной цепочки (можно считать его 0)
2. После распознавания каждого терминала указатель увеличивается на 1

В этом случае ВСЕГДА:

1. При входе в процедуру, распознающую любую конструкцию А, указатель будет установлен на первый символ подцепочки, порождаемой этой конструкцией А
2. При выходе из процедуры, распознающей конструкцию А, указатель будет установлен на символ, следующий за той подцепочкой, которая была порождена этой конструкцией А

Пример: трансляция представления BDD



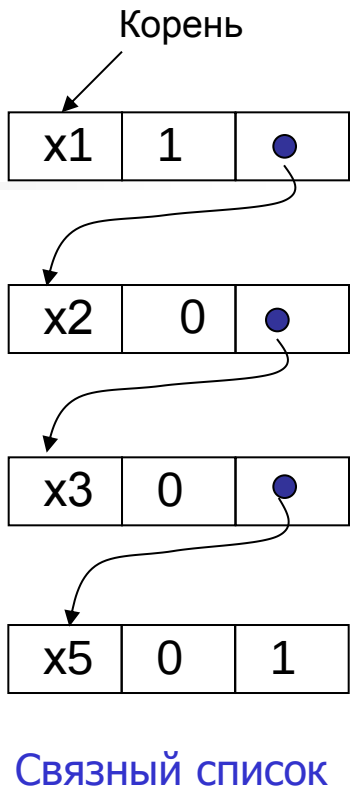
Формула БФ $f = \neg x_1 \vee x_2 x_3 x_5$

Каждый подграф BDD: v_k: [$x_j \rightarrow$ p0, p1]

p0 и p1 – подграфы, в которые из вершины v_k идут ребра, помеченные 0 и 1

Текстовое представление BDD примера:

BDD: { v5: [x1→1,v4]; v4: [x2→0,v3];
v3: [x3→0, v2] ; v2: [x5 →0, 1] }

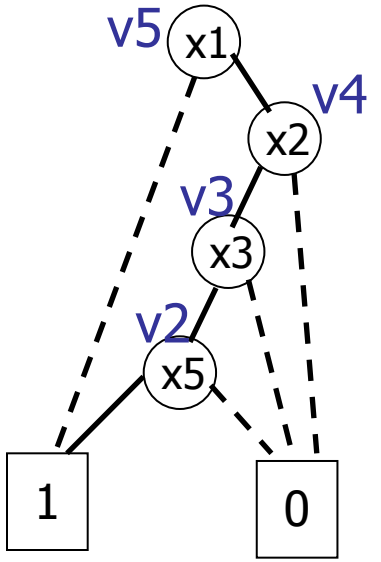


Связный список

Задача: построить транслятор из текстового представления BDD в связный список или в программу, вычисляющую значение функции f по значениям аргументов

Пример: трансляция представления BDD

BDD



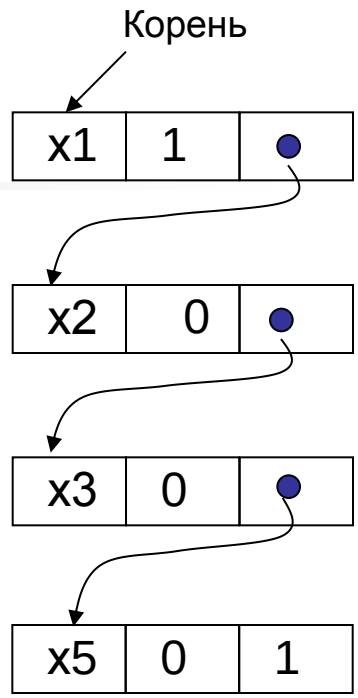
Формула БФ $f = \neg x_1 \vee x_2 x_3 x_5$

Каждый подграф BDD: v_k: [$x_j \rightarrow$ p0, p1]

p0 и p1 – подграфы, в которые из вершины v_k идут ребра, помеченные 0 и 1

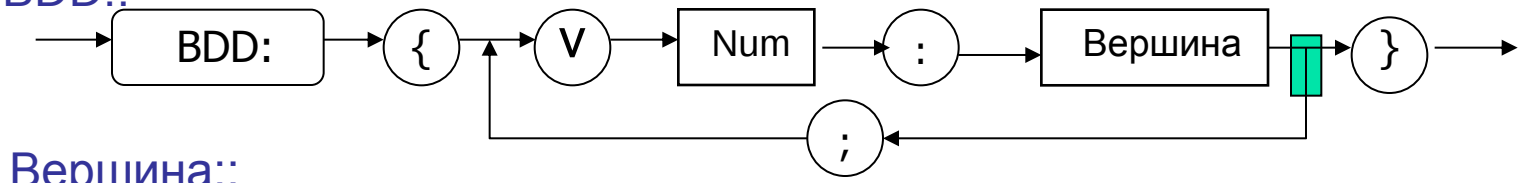
Текстовое представление BDD примера:

BDD: { v5: [x1→1,v4]; v4: [x2→0,v3];
v3: [x3→0, v2] ; v2: [x5 →0, 1] }



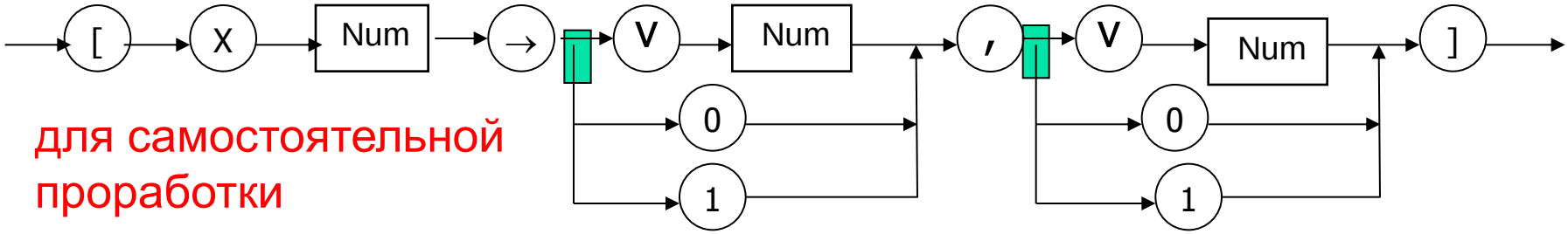
Связный список

BDD::



Грамматика текстового представления BDD

Вершина::



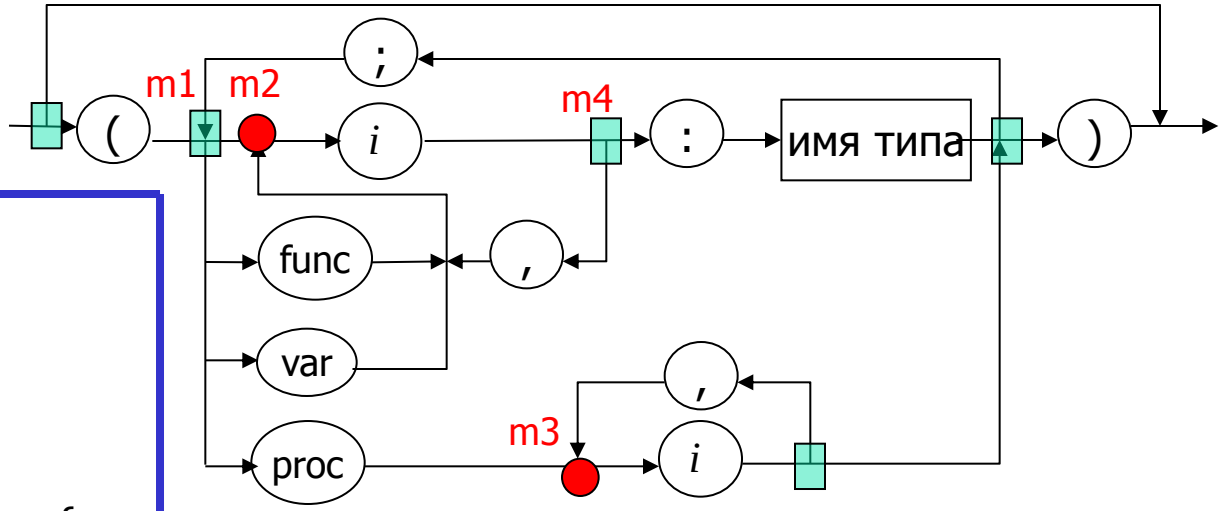
для самостоятельной проработки

Выводы: Транслятор рекурсивного спуска и LL(1) грамматики

- Транслятор рекурсивного спуска может строиться на любом языке программирования, в котором реализован рекурсивный вызов процедур
- При восстановлении синтаксического дерева при нисходящем разборе слева направо последовательно происходит восстановление дерева с корнем, помеченным левым очередным нетерминалом. Выбор проверяемого правила для текущего нетерминала определяется на основе анализа одного очередного терминала входной цепочки
- Это соответствует иерархическим вызовам процедур в программе, которые следуют друг за другом в охватывающей их процедуре. Для всех нетерминалов строятся соответствующие им процедуры, внутри которых вызовы других процедур – (нетерминалов) и проверки терминальных символов происходят в последовательности, определяемой их расположением в правилах грамматики
- Вызов процедуры или функции сопровождается занесением локальных параметров в стек, который поддерживается системными средствами. Стек соответствует магазину МП-автомата при разборе классическим LL(1) парзером
- Использование рекурсивного спуска позволяет быстро и наглядно писать программу распознавателя на основе грамматики. Главное, чтобы грамматика соответствовала виду LL(1)

Обнаружение ошибок: если очередной символ на входе не совпадает с терминалами в синтаксической диаграмме

List_of_par::

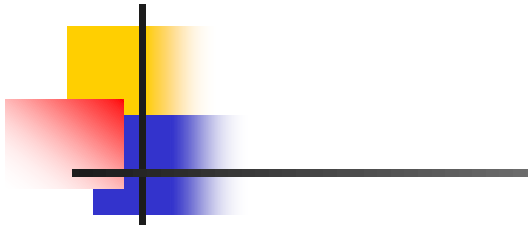



```
List_of_par( ) {
    if( s[Ук] == '(' ) then Ук++
     else { return };
m1: if( s[Ук] == 'i' ) then {
          Ук++; goto m4};
      else if( s[Ук] == 'func' ) then {
          Ук++; goto m2};
      else if( s[Ук] == 'var' ) then {
          Ук++; goto m2};
      else if( s[Ук] == 'proc' ) then {
          Ук++; goto m3};
       else { ERROR1( ) };
m2: if(s[Ук] == 'i' ) then {
          Ук++; goto m4 };
       else { ERROR2( ) } ;
...
    };
```

ERROR1( ): "*В списке параметров могут быть только переменные, функции (func), описания (var) и процедуры (proc), а у Вас – s[Ук] ...*"

ERROR2( ): "*После func или var в списке параметров ожидаем имя переменной, а у Вас – s[Ук] ...*"

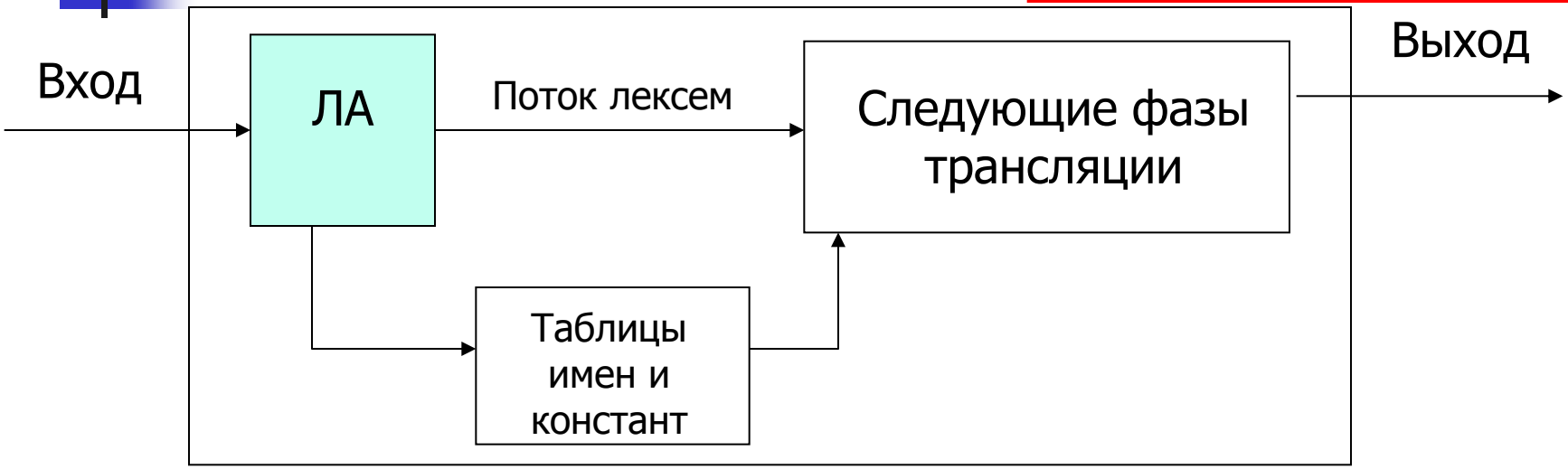
Существуют стратегии продолжения СА после выявления ошибки для того, чтобы найти большее число ошибок за проход



Построение компилятора языка Milan (Mini Language)

Сканер – первый проход транслятора ЯВУ

Лексема: | тип | номер |



begin i_0 ass read sc i_1 ass read sc while i_1 q_1 i_1 do if i_0 q_2 i_1 then i_0 ass i_0 aop_0 i_1 else i_1 ass i_1 aop_0 i_0 sc write lb i_1 rb end - **ЛА: программа ->поток лексем**

- Структура таблицы имен:

N	имя	тип
0	m13	int
1	alpha237	int
2		

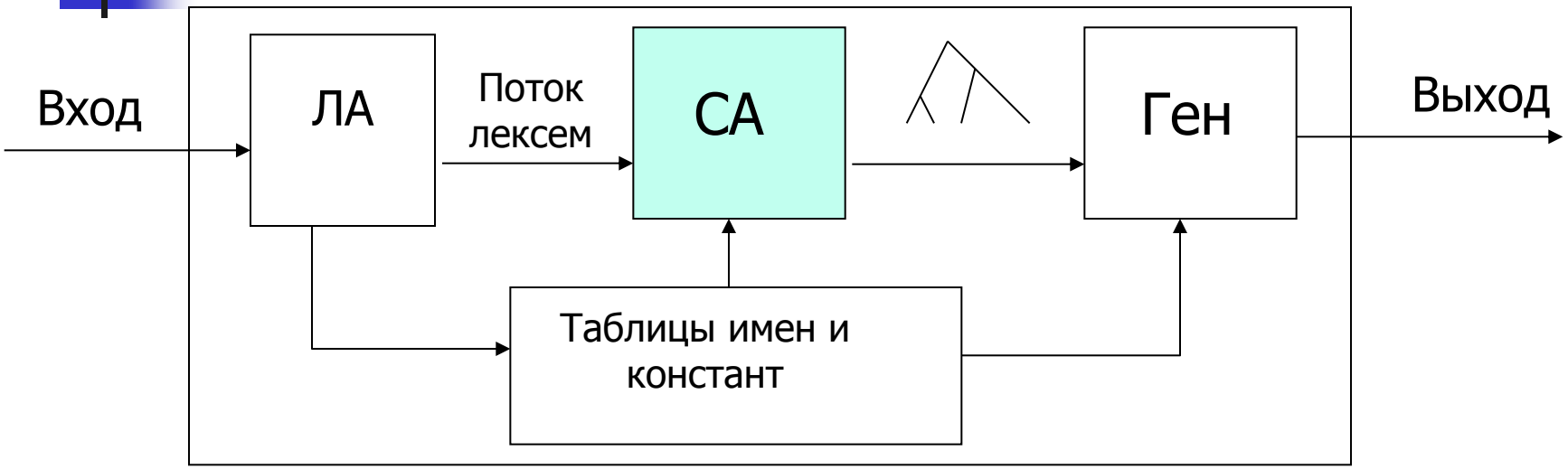
- Структура таблицы констант:

N	константа	значение
0	236	000101...
1	-43268	101100...
2		

Почему сканер выделяется в отдельный проход

- В принципе, задачи, решаемые сканером, можно возложить на синтаксический анализатор. Но такой подход неудобен по следующим причинам:
  - Использование лексем делает внутренне представление программы более удобным для дальнейшего анализа распознавателем. Последний манипулирует не отдельными символами, а законченными элементарными конструкциями, имеющими конкретное смысловое значение, что облегчает их общее восприятие и дальнейший семантический анализ
  - При построении лексем может осуществляться простейшая семантическая обработка. Например, преобразование и проверка числовых констант
  - Уменьшается длина программы, поступающей в синтаксический анализатор, за счет устранения из нее несущественных для дальнейшего анализа пробелов, комментариев, игнорируемых символов
  - Лексический анализатор использует более простые, по сравнению с синтаксическим анализатором, методы разбора

Полная схема компилятора Милана



Лексический анализ языка Милан рассмотрен

Наша задача: рассмотреть

- Синтаксический анализатор
- Генератор

Будем строить компилятор Милана методом рекурсивного спуска.

Если сумеем построить, то грамматика Милана недвусмысленна!

Грамматика языка Милан как цепочек лексем

```
begin
  a :=5;
  x:= read;
  a :=x+a*3;
  while x>0 do a:=a*2 od;
  x:=x-1
end
```

q_k = k=0? ‘=‘
k=1? ‘≠’
k=2? ‘<‘
k=3? ‘>’
k=4? ‘≤’
k=5? ‘≥’

$addop_k$ = (k=0)? ‘+‘: ‘-’

$mulop_k$ = (k=0)? ‘*’ : ‘/’

i_k: k - номер переменной

c_k: k - номер константы

Prog→begin L end
L →S | L ; S
S → i_k := E
| while B do L od
| if B then L fi
| if B then L else L fi
| write (E)
B→ E q_k E
E→ E $addop_k$ T | T
T→ T $mulop_k$ P | P
P→ i_k | c_k | (E) | read

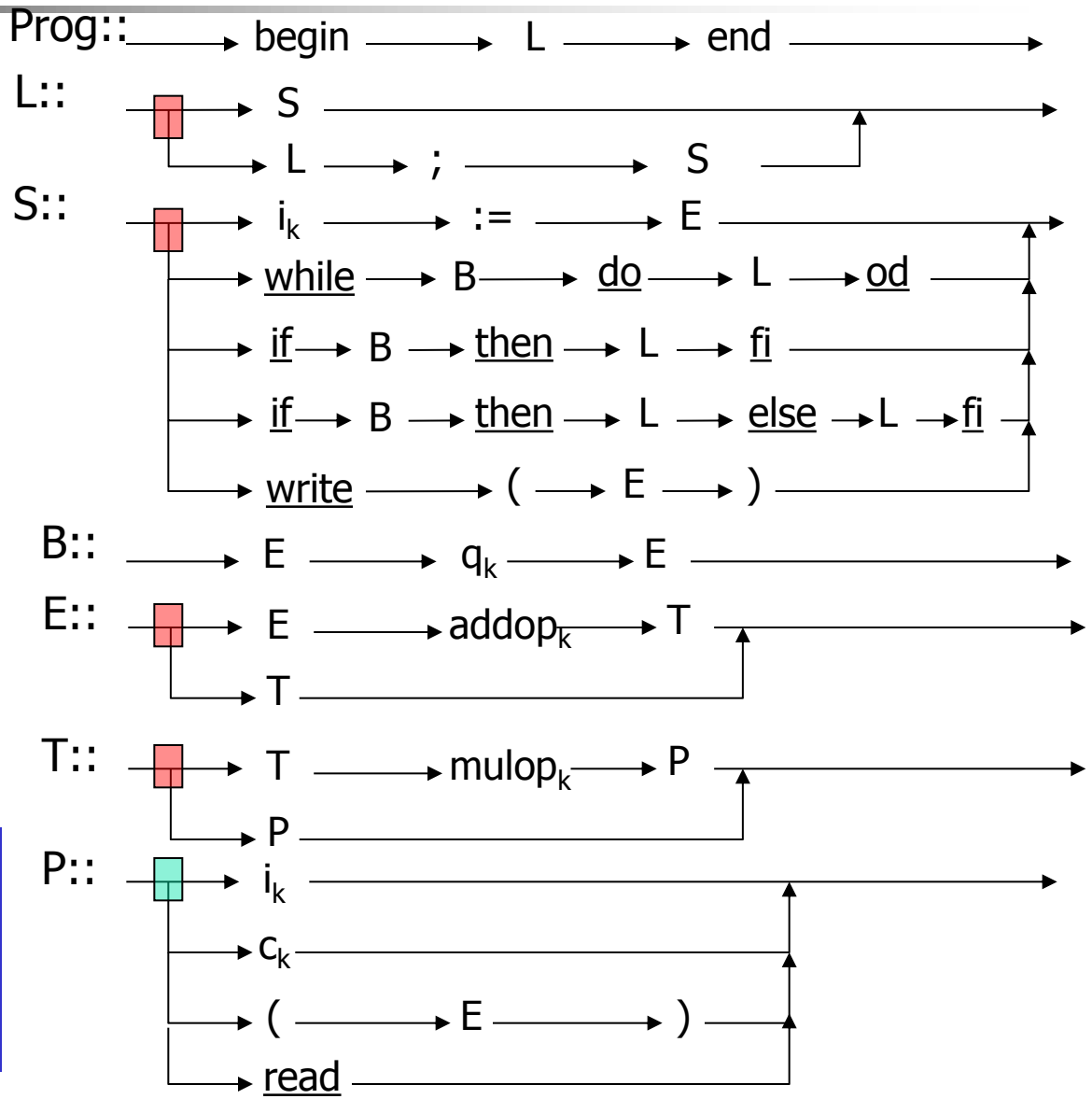
В грамматике Милана:

1. Убрана неоднозначность введением закрывающих скобок od, fi, ... и специальным заданием грмматики арифметических выражений
2. Грамматика строится для потока лексем, в котором одной лексемой с номером представлены:
   все имена,
   все константы,
   каждое служебное слово (begin, while, ...)
   все операции отношения,
   арифметические операции {+, -} и {*, /}

Стала ли грамматика однозначной?

Синтаксические диаграммы для грамматики Милана

Prog→begin L end
L →S | L ; S
S → i_k := E
| while B do L od
| if B then L fi
| if B then L else L fi
| write (E)
B→ E q_k E
E→ E $addop_k$ T | T
T→ T $mulop_k$ P | P
P→ i_k | c_k | (E) | read



Синтаксические диаграммы для L, S, E и T – не удовлетворяют правилу рекурсивного спуска

Убираем левую рекурсию в правилах для L

Prog→begin L end

L →S | L ; S S { ; S }

S → i_k := E

| while B do L od

| if B then L fi

| if B then L else L fi

| write (E)

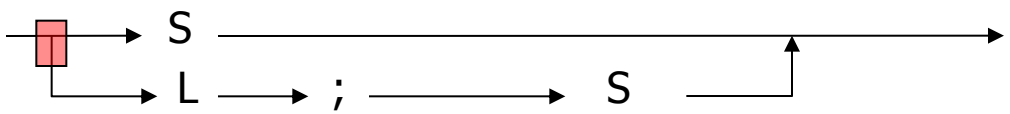
B→ E q_k E

E→ E $addop_k$ T | T

T→ T $mulop_k$ P | P

P→ i_k | c_k | (E) | read

L::



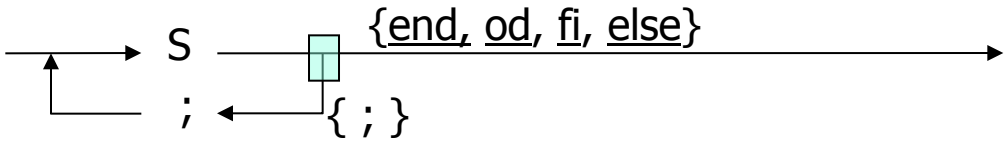
Вместо

L → S | L ; S

эквивалентное правило

L ::= S { ; S }

L::



Синтаксические диаграммы для L, S, E и T – не удовлетворяют правилу рекурсивного спуска

Факторизация правил для S: условный оператор

Prog→begin L end

L →S | L ; S

S → i_k := E

| while B do L od

| if B then L fi

| if B then L else L fi

| write (E)

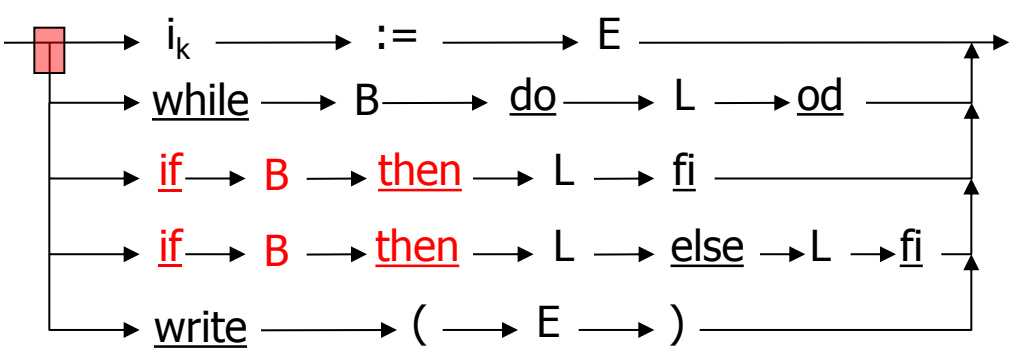
B→ E q_k E

E→ E $addop_k$ T | T

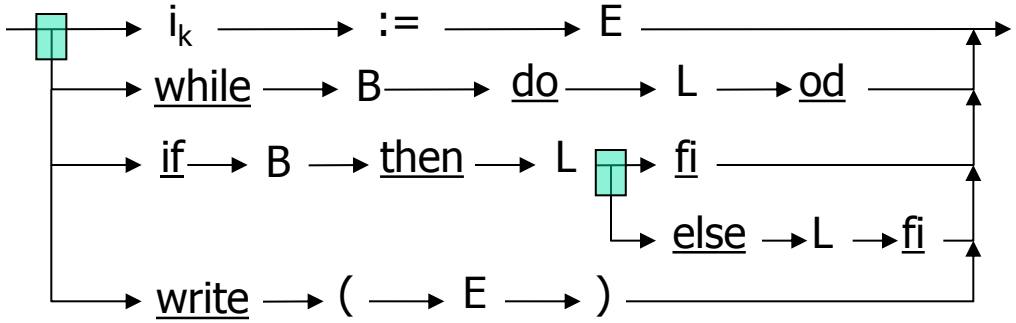
T→ T $mulop_k$ P | P

P→ i_k | c_k | (E) | read

S::



S::



Синтаксические диаграммы для L, S, E и T – не удовлетворяют правилу рекурсивного спуска

Убираем левую рекурсию в правилах для E

Prog→<u>begin</u> L <u>end</u>

L →S | L ; S

S → i_k := E

| <u>while</u> B <u>do</u> L <u>od</u>

| <u>if</u> B <u>then</u> L <u>fi</u>

| <u>if</u> B <u>then</u> L <u>else</u> L <u>fi</u>

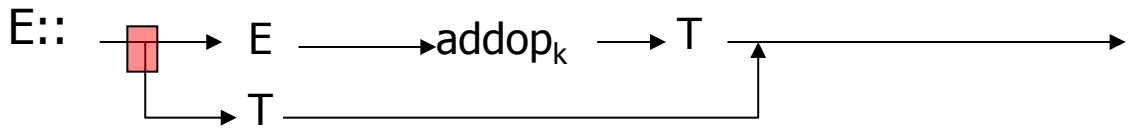
| <u>write</u> (E)

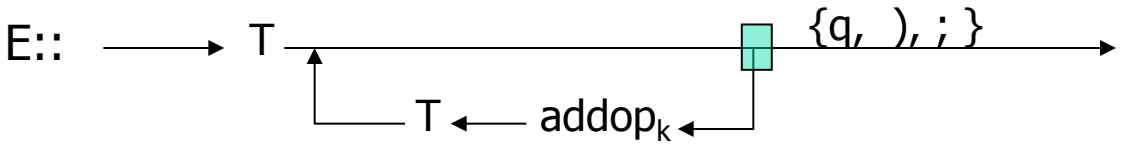
B→ E q_k E

E→ E $addop_k$ T | T T {$addop_k$ T}

T→ T $mulop_k$ P | P

P→ i_k | c_k | (E) | <u>read</u>





Синтаксические диаграммы для L, S, E и T – не удовлетворяют правилу рекурсивного спуска

Убираем левую рекурсию в правилах для T

Prog→<u>begin</u> L <u>end</u>

L →S | L ; S

S → i_k := E

| <u>while</u> B <u>do</u> L <u>od</u>

| <u>if</u> B <u>then</u> L <u>fi</u>

| <u>if</u> B <u>then</u> L <u>else</u> L <u>fi</u>

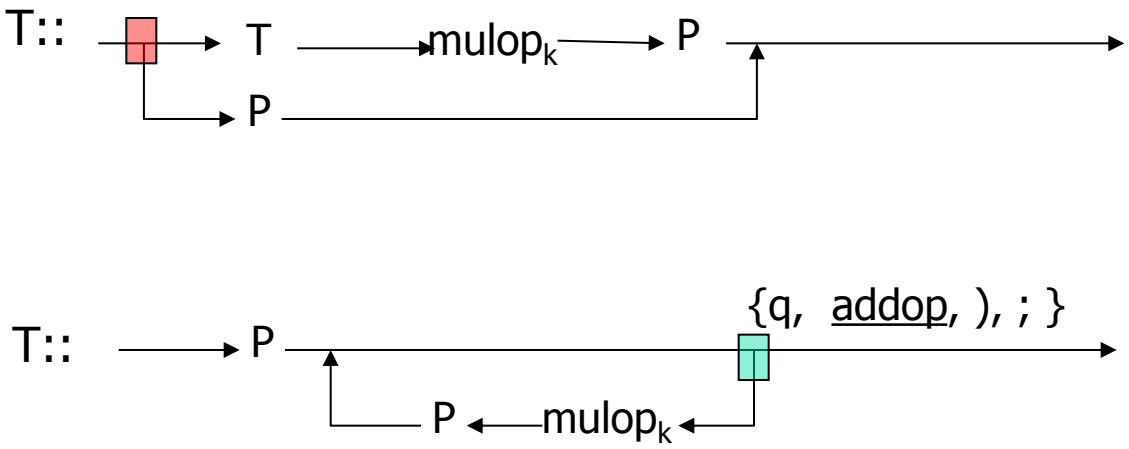
| <u>write</u> (E)

B→ E q_k E

E→ E $addop_k$ T | T

T→ T $mulop_k$ P | P P {mop_k P}

P→ i_k | c_k | (E) | <u>read</u>



Синтаксические диаграммы для L, S, E и T – не удовлетворяют правилу рекурсивного спуска

Диаграммы рекурсивного спуска для языка Милан

Prog→begin L end

L →S | L ; S L ::= S { ; S }

S → i_k := E

| while B do L od

| if B then L fi

| if B then L else L fi

| write (E)

B→ E q_k E

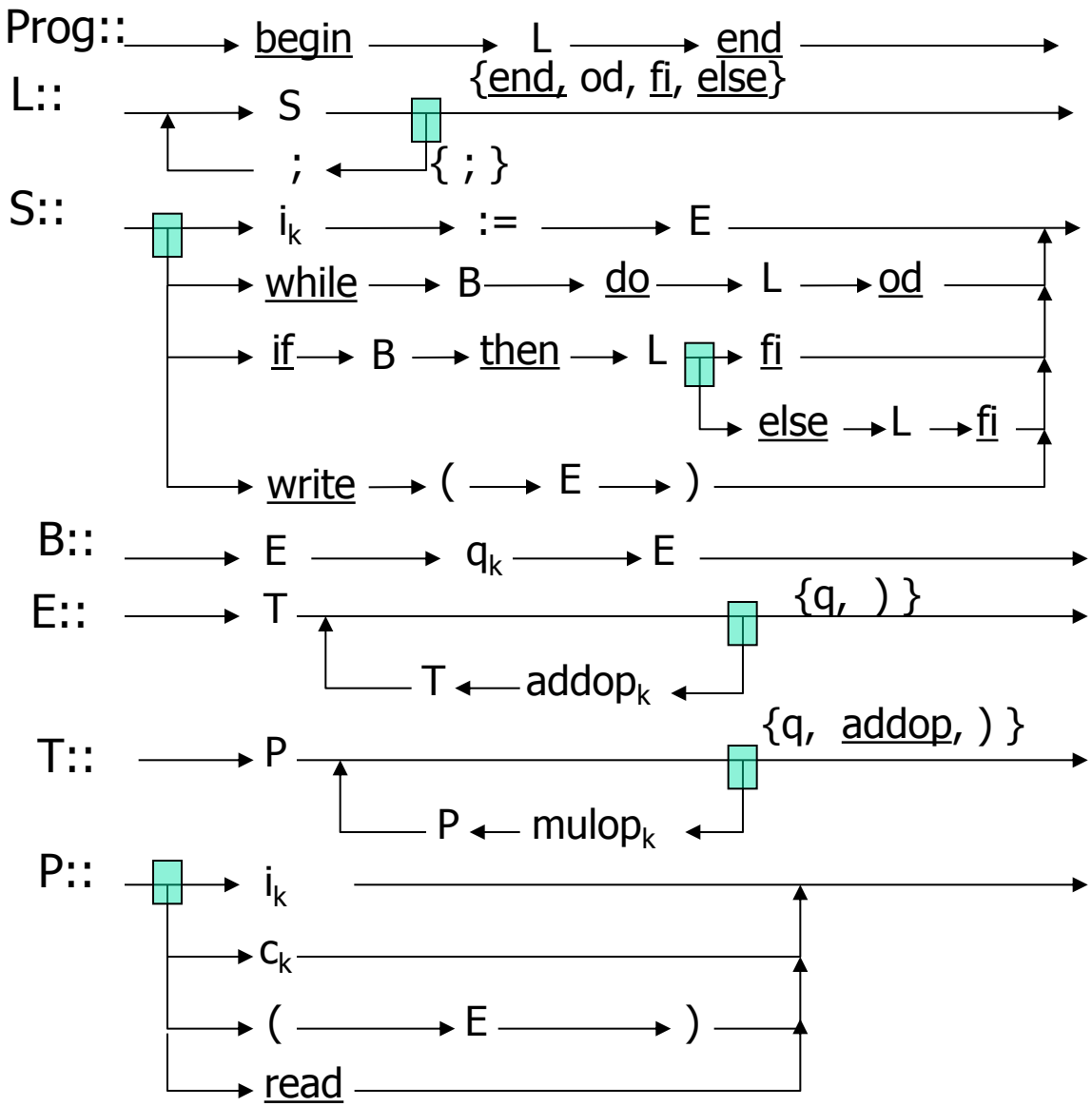
E→ E $addop_k$ T | T E::=T{$addop_k$ T}

T→ T $mulop_k$ P| P T::=P{$mulop_k$ P}

P→ i_k | c_k | (E) | read

Убираем левую рекурсию и факторизуем

Синтаксические диаграммы удовлетворяют требованиям однозначного выбора пути

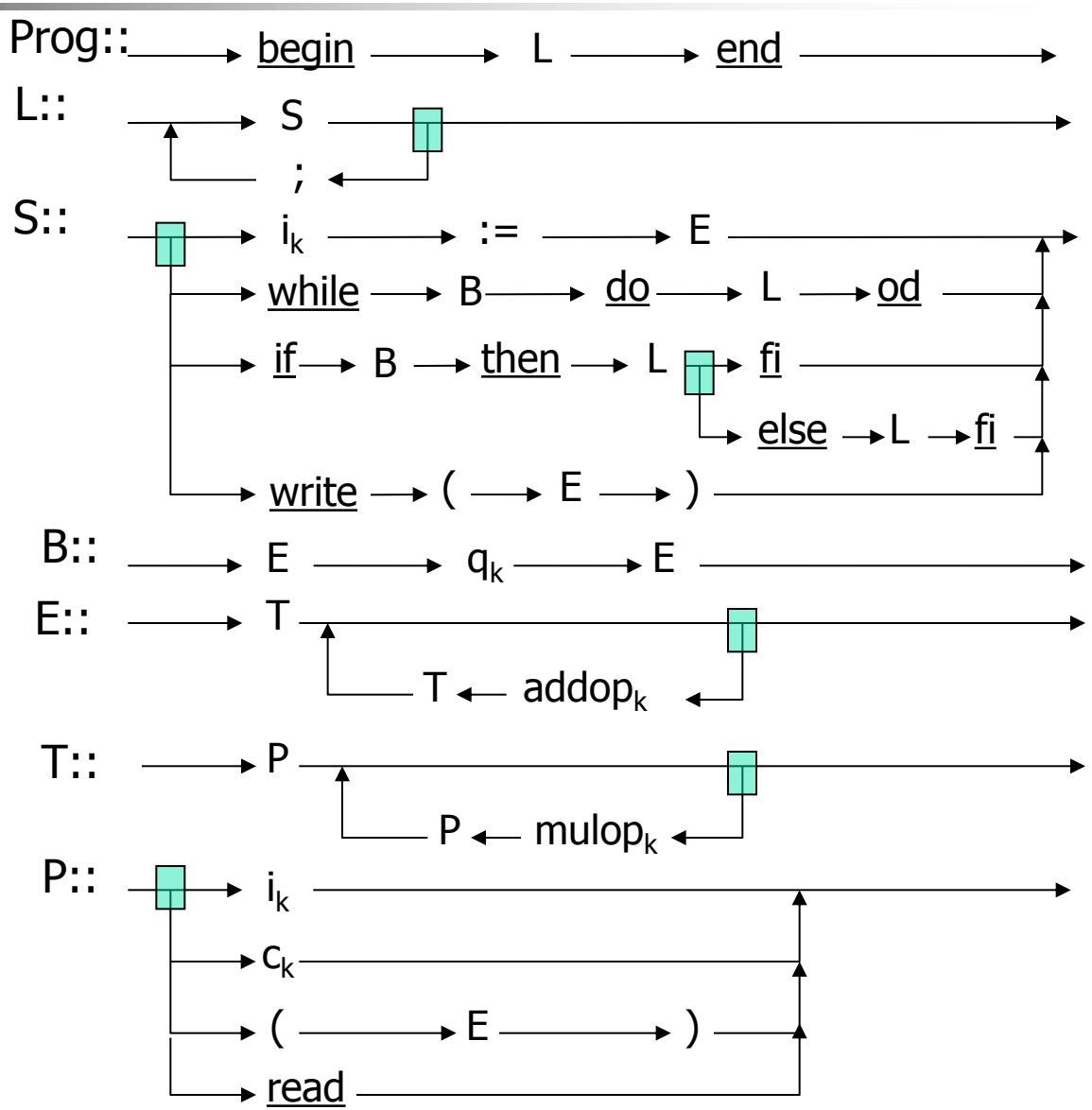


Процедуры рекурсивного спуска для языка Милан строятся однозначно

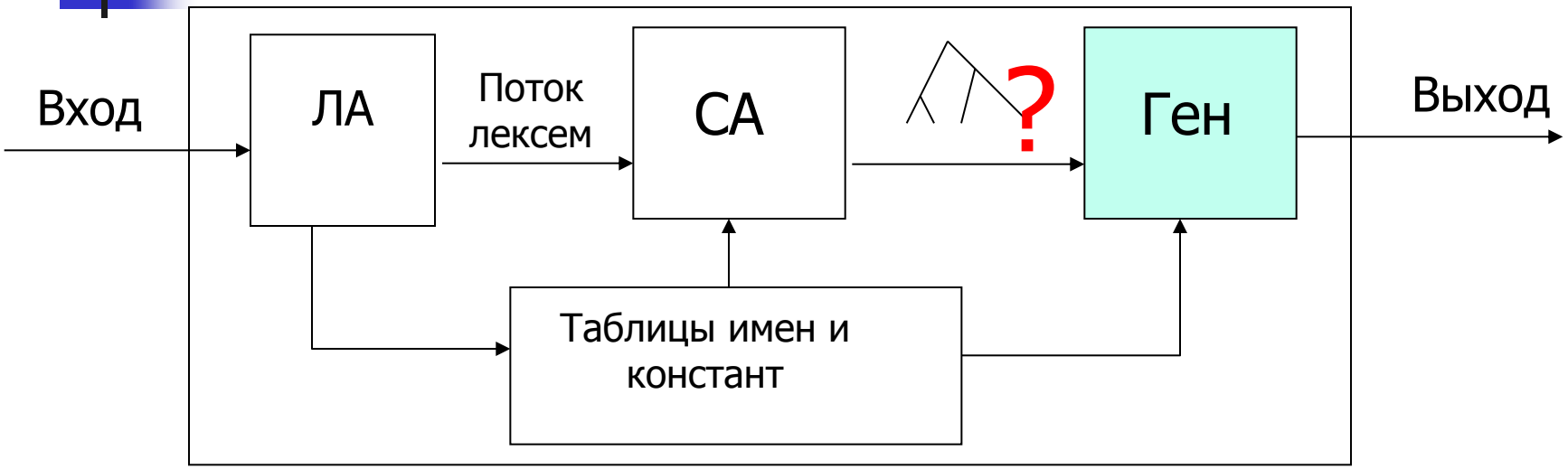
```
int Ук:=0; // указатель

main( ) {
    if( s[Ук] != 'begin') then {
       Error1( )
    }
 else { Ук++;
        L ( );
        if( s[Ук] == 'end') then {
       Ук++;  return }
       else { Error2( ) };
     }
};

L ( ) {
 m: S( );
     f( s[Ук] == ';') then { Ук++;
     goto m };
};
S ( ) {
 ...
```



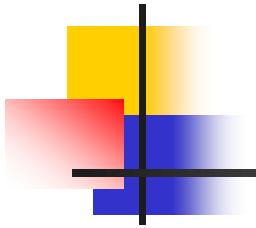
Полная схема компилятора Милана



При синтаксическом анализе методом рекурсивного спуска структура входной строки отражается неявно в последовательности вызовов модулей, соответствующих нетерминалам грамматики

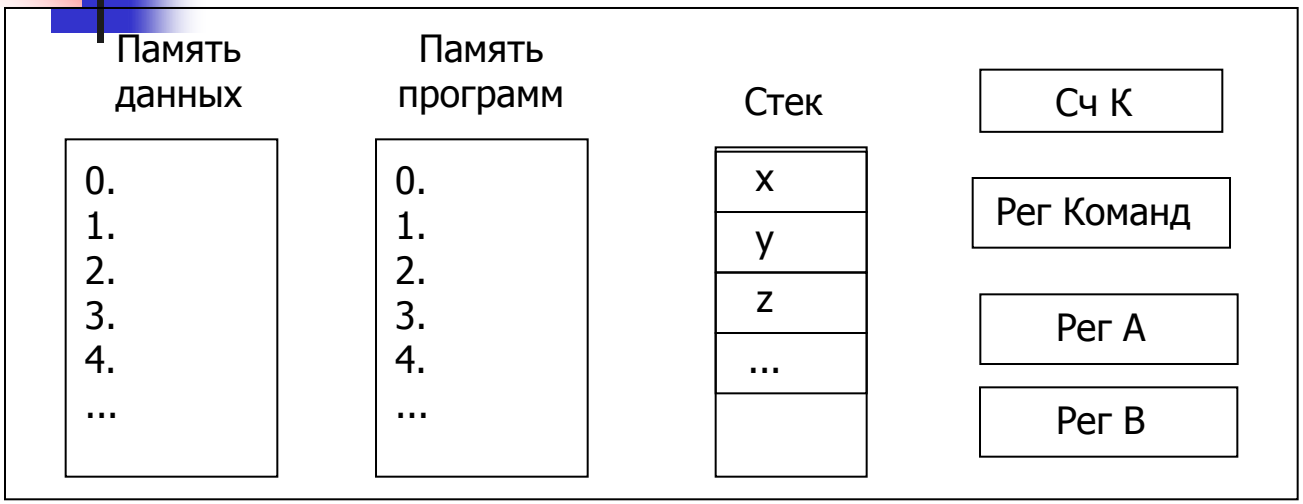
При выполнении этих процедур можно выполнять и семантические операции

Реализуем компиляцию в р-код стековой машины



Целевая машина – p-код

Целевая машина – p-код (стековая машина)

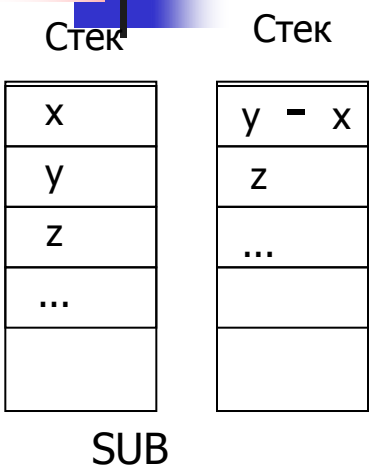


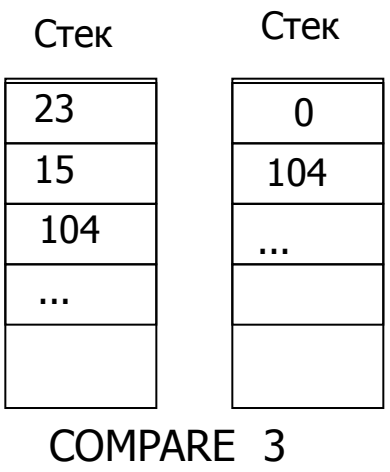
- Арифметика:
  ADD
  SUB
  MULT
  DIV
- Пересылки
  LOAD Addr a
  STORE Addr a
- PUSH const
- Сравнение:
  COMPARE k

q_k = k=0? '='
k=1? '≠'
k=2? '<'
k=3? '>'
k=4? '≤'
k=5? '≥'

Пример выполнения стековых команд

$a - b \le c + d \times 5$





15 > 23 ?

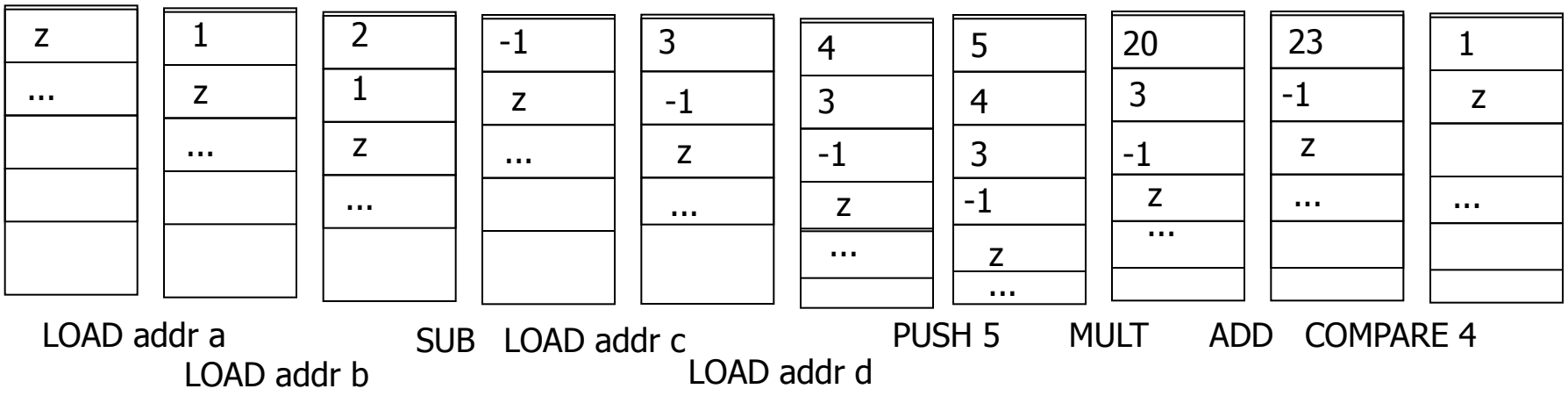
q_k = k=0? '='
k=1? '≠'
k=2? '<'
k=3? '>'
k=4? '≤'
k=5? '≥'

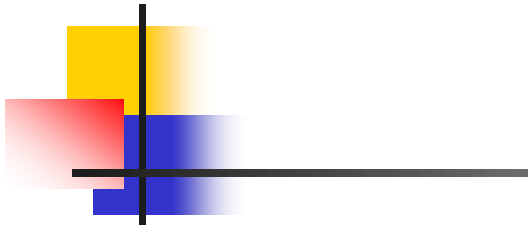
```
LOAD    addr a
LOAD    addr b
SUB
LOAD    addr c
LOAD    addr d
PUSH    5
MULT
ADD
COMPARE 4
```

$a - b \le c + d \times 5$?

пусть a=1, b=2, c=3, d=4

$a - b \le c + d \times 5$ да!!





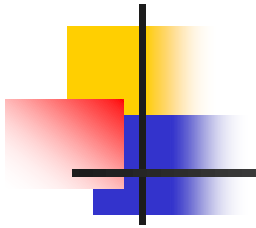
- Обработка выражений языка Милан

Пример 1 генерации p-кода: что нужно получить

```
begin
    m:=read;  n:=read;
    k:=m-n;   n:=k+m*7;
    write (m*n)
end
```

- Арифметика:
  - ADD
  - SUB
  - MULT
  - DIV
- Пересылки
  - LOAD addr a
  - STORE addr a
  - PUSH const
- Команды интерфейса:
  - INPUT – из входного потока – в верхушку стека
  - PRINT – из верхушки стека – на печать
  - STOP – возврат управления ОС

```
 0:  INPUT                 } m:=read
 1:  STORE    addr m       }
 2:  INPUT                 } n:=read
 3:  STORE    addr n       }
 4:  LOAD     addr m       } k:=m-n
 5:  LOAD     addr n       }
 6:  SUB                   }
 7:  STORE    addr k       }
 8:  LOAD     addr k       } n:=k+m*7
 9:  LOAD     addr m       }
10:  PUSH     7            }
11:  MULT                  }
12:  ADD                   }
13:  STORE    addr n       }
14:  LOAD     addr m       } write(m*n)
15:  LOAD     addr n       }
16:  MULT                  }
17:  PRINT                 }
18:  STOP
```

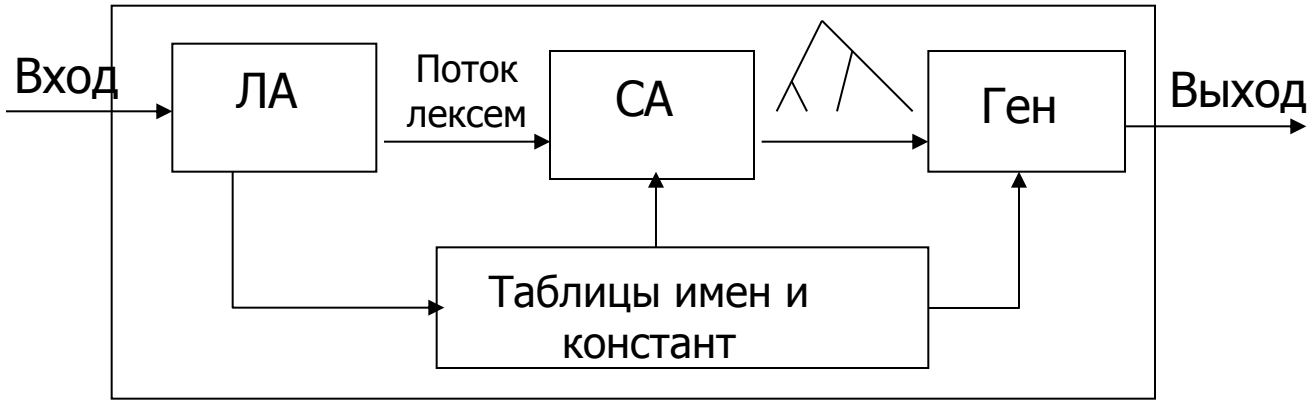


Как построить компилятор с языка Милан на язык стековой машины???

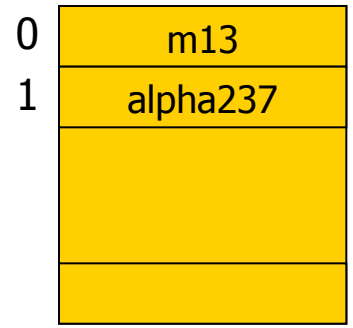
В синтаксические диаграммы вставим семантические действия: генерацию нужных команд

Наша задача – встроить такие семантические операции в распознаватель рекурсивного спуска, чтобы для любой входной программы на языке Милан генерировался правильный код

Откуда берутся адреса переменных и констант



Память данных стековой машины



Структура таблицы имен:

N	имя	тип
0	m13	int
1	alpha237	int
2		

Структура таблицы констант:

N	константа	значение
0	236	000101...
1	-43268	101100...
2		

- Будем считать, что после лексического анализа транслятор распределяет память стековой машины так: первые регистры памяти данных отводятся для переменных в порядке их встречаемости в таблице имен

С указывает на очередную команду в будущей программе

Основной слайд!!

Обработка выражений языка Милан

y_0: C:=0; y_{20}: Gen(C: STOP)

Основная идея:
результатом обработки каждого подвыражения арифметического выражения является последовательность команд, при выполнении которых в ВЕРХУШКУ стека помещается результат вычисления этого подвыражения

y_1: Gen(C: LOAD k); C++;

y_2: Gen(C: PUSH c_k); C++;

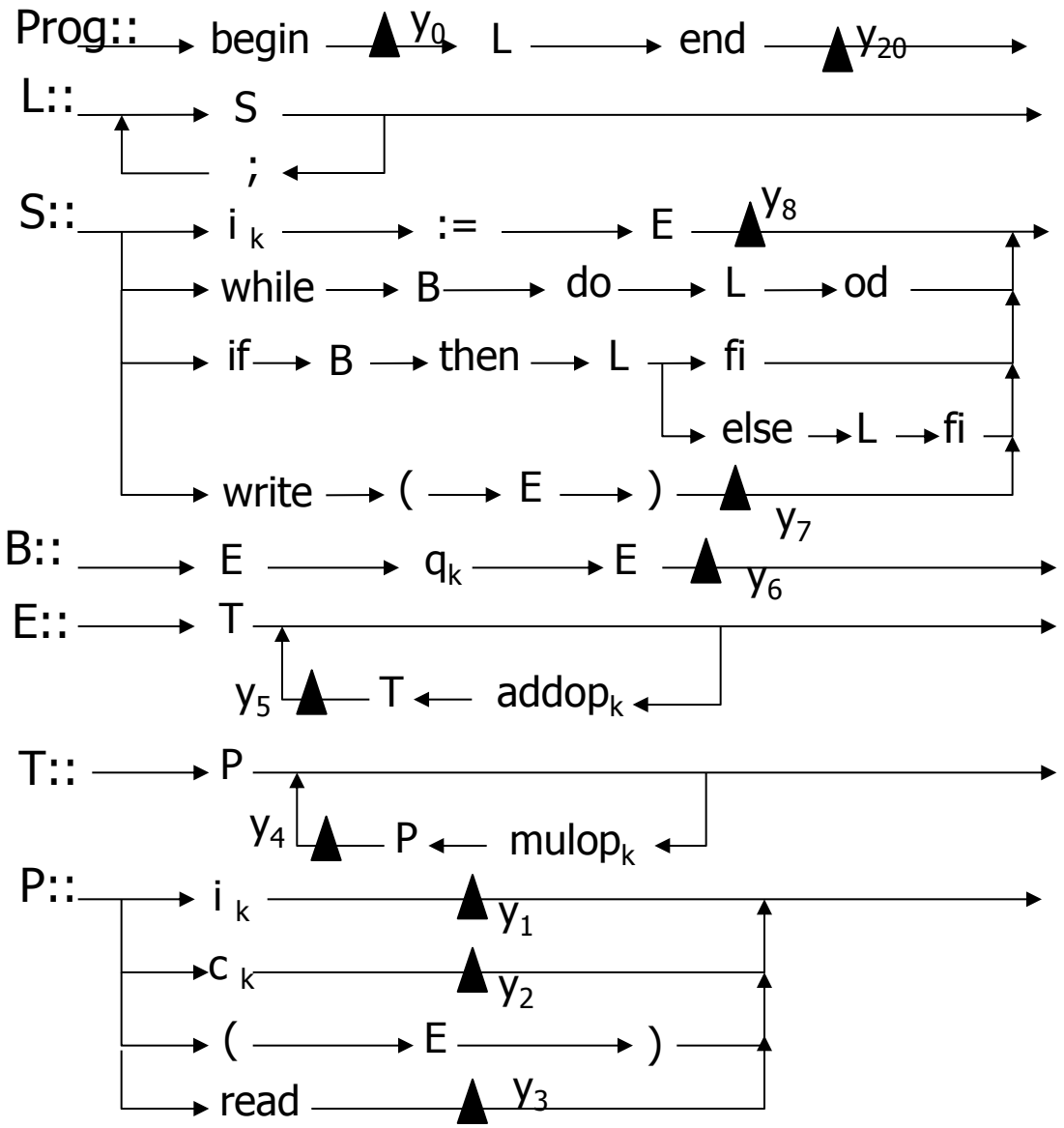
y_3: Gen(C: INPUT); C++;

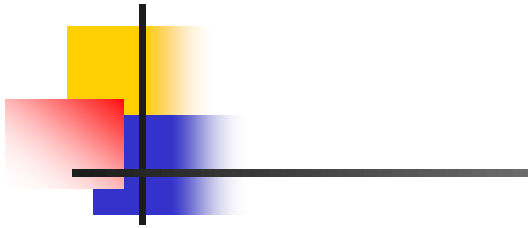
y_4: k=0? Gen(C: MUL) :
Gen(C: DIV); C++;

y_6: Gen(C: COMPARE k); C++;

y_7: Gen(C: OUTPUT); C++;

y_8: Gen(C: STORE k); C++;



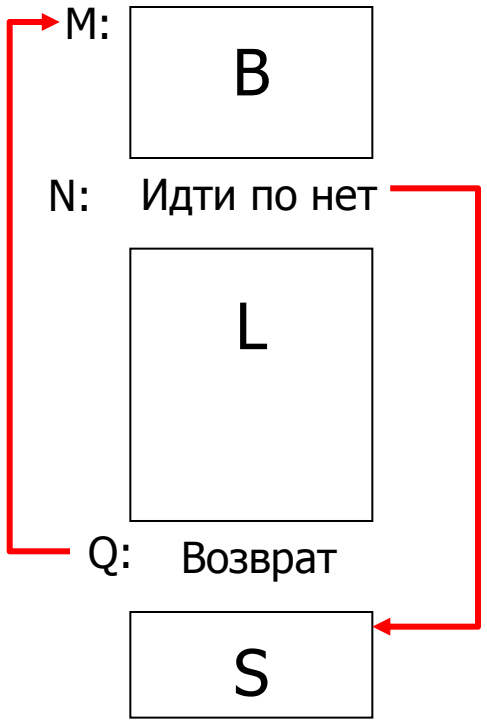


Обработка управляющих конструкций языка Милан

Реализация (семантика) оператора цикла

while B do L od; S

Программа стековой машины



Команды, вычисляющие B

Команды, реализующие L

Команды, реализующие S

Для реализации цикла нужно вставить команды перехода в нужные места

Команды стековой машины, вычисляющие B, помещают в вершину стека либо 0, либо 1 в зависимости от истинности условного выражения B

Пример 2 генерации р-кода для “while B do L od; S”

```
begin
    m:=read;
    n:=read;
    while m>n do
       m:=m*2;
       n:=n-1
    od
    write (m)
end
```

q_k = k=0? ‘=‘
k=1? ‘≠’
k=2? ‘<‘
k=3? ‘>’
k=4? ‘≤’
k=5? ‘≥’

```
0:  INPUT
1:  STORE   addr m
2:  INPUT
3:  STORE   addr n
4:  LOAD    addr m
5:  LOAD    addr n
6:  COMPARE  3
7:  JUMP_NO  17
8:  LOAD    addr m
9:  PUSH    2
10: MULT
11: STORE   addr m
12: LOAD    addr n
13: PUSH    1
14: SUB
15: STORE   addr n
16: JUMP     04
17: LOAD    addr m
18: PRINT
19: STOP
```

m:=read

n:=read

B

если NO

m:=m*2

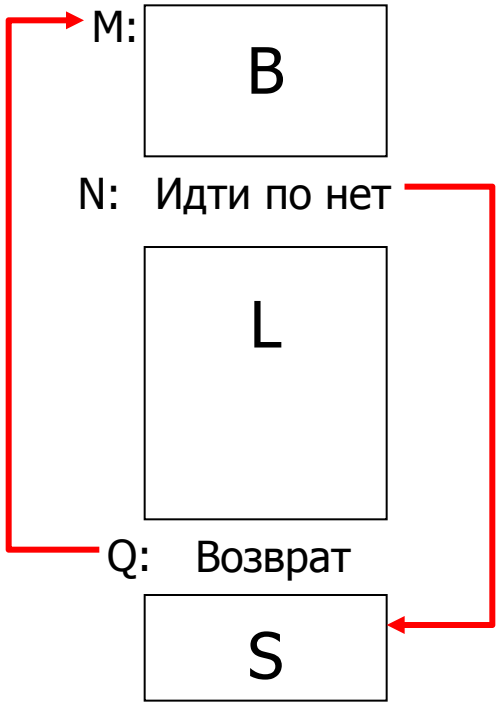
n:=n-1

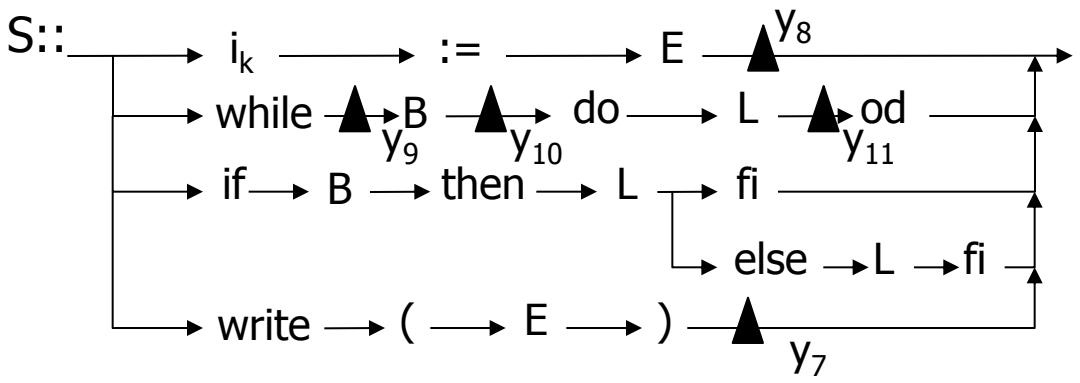
write(m)

- Команды перехода и сравнения:
  - COMPARE k – два верхних элемента стека сравниваются (отношение k). Результат (TRUE или FALSE) помещается в стек
  - JUMP k – переход на команду с адресом k
  - JUMP_NO k – переход на команду с адресом k, если в верхушке стека FALSE (верхушка стека выбрасывается)
  - JUMP_YES k – переход на команду с адресом k, если в верхушке стека TRUE (верхушка стека выбрасывается)

Генерация команд для организации цикла while-do

while B do L od; S





y_9: запомнить в M адрес начала блока B

y_{10}: запомнить текущий адрес в N и увеличить счетчик-оставить место для команды перехода

y_{11}: построить две команды:
в C – JUMP команду возврата на M (Q указывает счетчик C)
в N – JUMP_NO – команду перехода на следующую команду

y_9: M:= C;

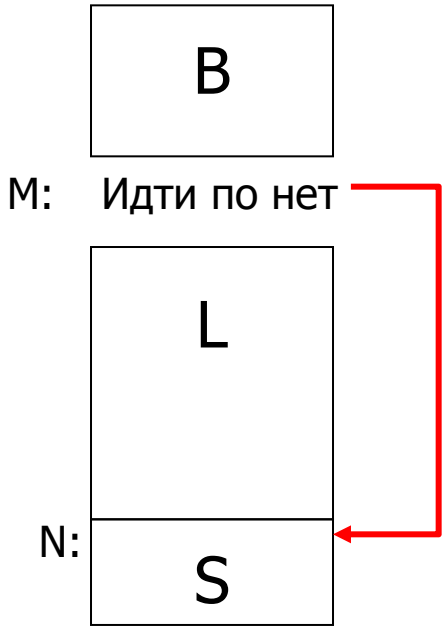
y_{10}: N:= C; C++;

y_{11}: Gen(C: JUMP M); C++; Gen(N: JUMP_NO C)

Реализация (семантика) условного оператора

if B then L fi; S

Программа стековой машины



Пример 3 генерации р-кода для "if B then L fi; S"

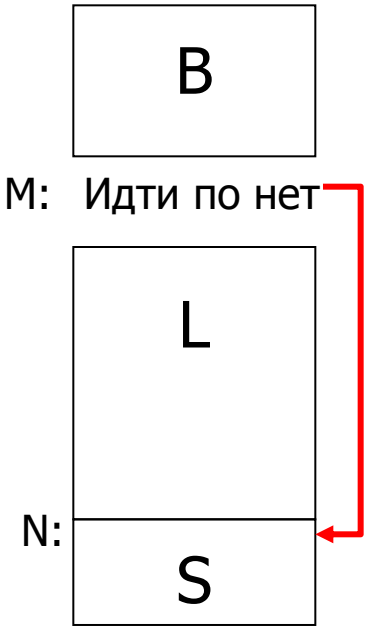
```
begin
    m:=read;
    n:=read;
    if m>n then
       m:=m*2;
       n:=n-1
     fi;
    write (m)
      end
```

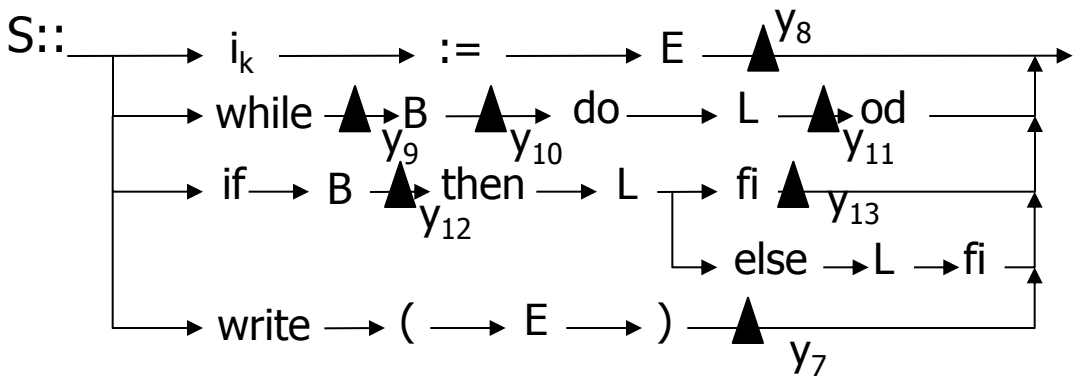
q_k = k=0? '='
k=1? '≠'
k=2? '<'
k=3? '>'
k=4? '≤'
k=5? '≥'

```
0:  INPUT                    } m:=read
1:  STORE    addr m
2:  INPUT                    } n:=read
3:  STORE    addr n
4:  LOAD     addr m          } B
5:  LOAD     addr n
6:  COMPARE  3
7:  JUMP_NO  16              если NO
8:  LOAD     addr m          } m:=m*2
9:  PUSH     2               } n:=n-1
10: MULT
11: STORE    addr m
12: LOAD     addr n
13: PUSH     1
14: SUB
15: STORE    addr n
16: LOAD     addr m          } write(m)
17: PRINT
18: STOP
```

Генерация команд для организации условного оператора

if B then L fi; S





y_{12}: запомнить в M место для команды
M:= C; C++

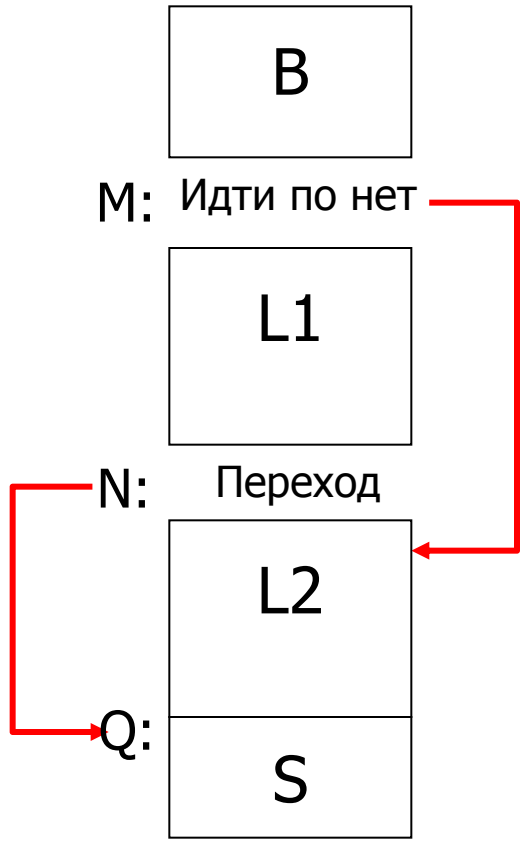
y_{13}: построить в M команду JUMP_NO перехода на C:
Gen(M: JUMP_NO C)

После того, как обработали все символы, относящиеся к L, счетчик команд C показывает N – начало блока команд очередного оператора

Реализация (семантика) условного оператора

if B then L1 else L2 fi; S

Программа стековой машины

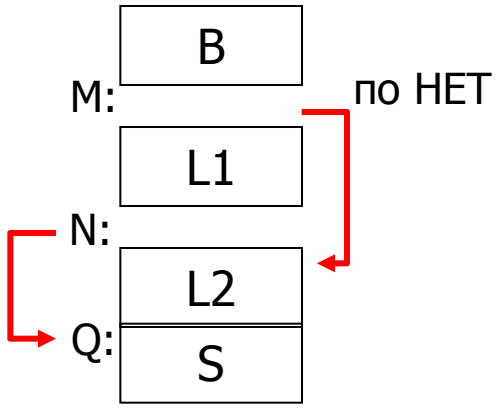


Пример 4 генерации р-кода для "if B then L1 else L2 fi; S"

```
begin
    m:=5;
    n:=read;
    if m≤n then
       m:=m*2
    else
       n:=n-1
     fi;
    write (m)
end
```

q_k = k=0? '='
k=1? '≠'
k=2? '<'
k=3? '>'
k=4? '≤'
k=5? '≥'

```
00  PUSH      5          } m:=5
01  STORE     addr m     }
02  INPUT                } n:=read
03  STORE     addr n     }
04  LOAD      addr m     }
05  LOAD      addr n     } B
06  COMPARE   4          }
07  JUMP_NO   13         если NO
08  LOAD      addr m     }
09  PUSH      2          } L1
10  MUL                  }
11  STORE     addr m     }
12  JUMP      17
13  LOAD      addr n     }
14  PUSH      1          } L2
15  SUB                  }
16  STORE     addr n     }
17  LOAD      addr m     } S
18  OUTPUT               }
19  STOP
```

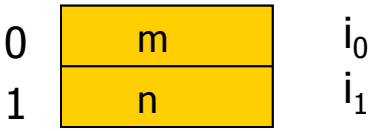


Пример 5

```
begin
    m:=5;
    n:=read*234;
    if m≥n then
       m:=m*2
    else
       while n>10 do
         n:=n-1
       od
     fi;
     write (m+3)
end
```

q_k = k=0? '='
k=1? '≠'
k=2? '<'
k=3? '>'
k=4? '≤'
k=5? '≥'

Память данных стековой машины




```
00  PUSH     5
01  STORE    0
02  INPUT
03  PUSH     234
04  MUL
05  STORE    1
06  LOAD     0
07  LOAD     1
08  COMPARE  5
09  JUMP_NO  15
10  LOAD     0
11  PUSH     2
12  MUL
13  STORE    0
14  JUMP     24
```

m:=5;
n:= read*234

m ≥ n?

m :=m*2

n>10?

```
15  LOAD     1
16  PUSH     10
17  COMPARE  3
18  JUMP_NO  24
```

n:=n-1

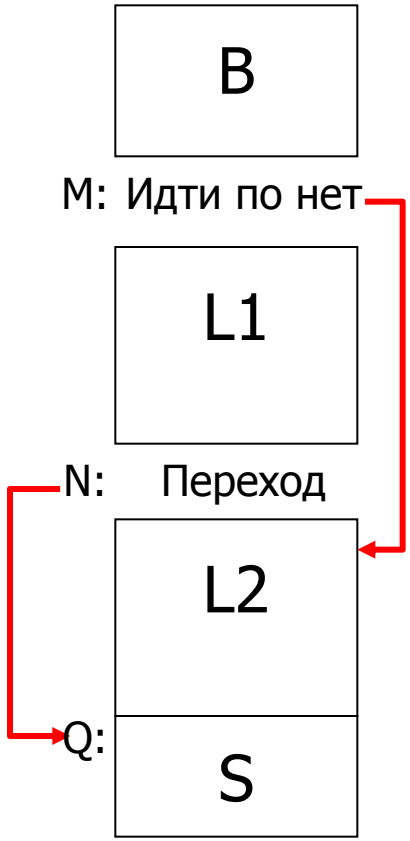
```
19  LOAD     1
20  PUSH     1
21  SUB
22  STORE    1
23  JUMP     15
```

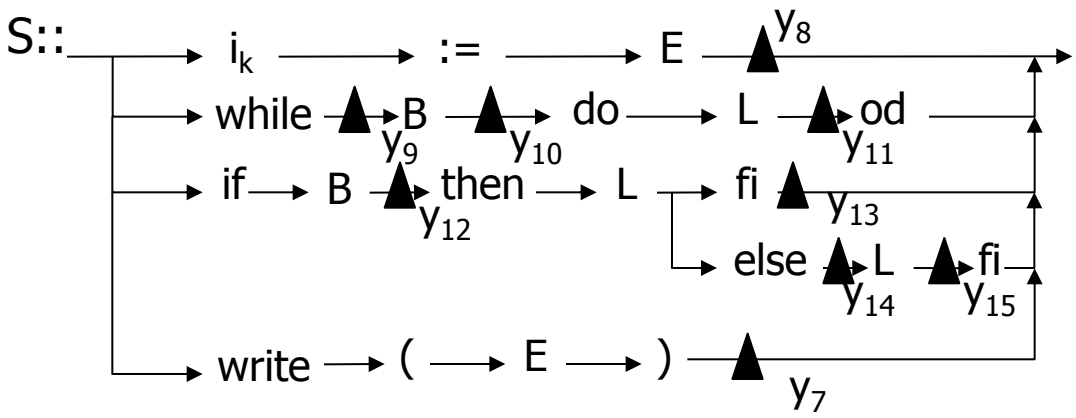
write (m+3)

```
24  LOAD     0
25  PUSH     3
26  ADD
27  OUTPUT
28  STOP
```

Генерация команд для организации условного оператора

if B then L1 else L2 fi; S





y_{12}: запомнить в M место для команды
M:= C; C++

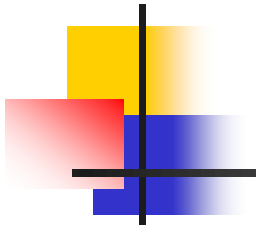
Было!

y14: запомнить в N адрес, куда потом поместим будущую команду перехода на команды, вычисляющие S
N:= C; C++

y15: сгенерировать две команды перехода:
Gen(M: JUMP_NO N+1);
Gen(N: JUMP C) Q показывает счетчик C

Курсовая работа по построению компилятора

- Транслятор базового языка Милан построен в точности в соответствии с представленными алгоритмами. Этот транслятор доступен всем
- В курсовой работе каждому предлагается внести в базовый компилятор изменения, необходимые для трансляции расширения языка Милан, у каждого – свое конкретное расширение
- Выполнение курсовой работы предполагает тщательное изучение теории построения трансляторов методом рекурсивного спуска, изучение программной реализации базового компилятора, внесение своих дополнений в лексический анализатор, синтаксический анализатор и семантики. Возможно изменение стековой машины (включение дополнительных нужных команд)
- Проверка выполнения курсовой работы состоит в проверке полного понимания базовой реализации (ее алгоритмов работы и самой программы) и своего расширения, которое должно работать на нескольких примерах и выдавать разумные объяснения синтаксическим ошибкам
- Для получения дополнительных баллов студенту требуется дополнительно построить компилятор, на базе производственного компилятора компиляторов Yacc (Yet another compiler compilers) или Bison, работающего на основе восходящего метода синтаксического анализа LR(k), к изучению которого мы приступаем со следующей лекции



Спасибо за внимание