ФГОБУ ВПО "СибГУТИ"
Кафедра вычислительных систем

ЯЗЫКИ ПРОГРАММИРОВАНИЯ /
ПРОГРАММИРОВАНИЕ

Модульное программирование

Преподаватель:

Доцент Кафедры ВС, к.т.н.

Поляков Артем Юрьевич

Императивная парадигма программирования

- Вычислительный процесс описывается в виде инструкций, изменяющих состояние программы.
- Аналогична приказам, выражаемым повелительным наклонением в естественных языках, то есть это последовательность команд, которые должен выполнить компьютер.
- Противопоставлена декларативной парадигме (предполагает описание результата, а не способ его получения).

Процедурное программирование

- Выполнение программы сводится к последовательному выполнению операторов с целью преобразования исходного состояния памяти, то есть значений исходных данных, в заключительное, то есть в результаты.
- Каждый шаг алгоритма однозначно определяется программистом.
- Задачи разбиваются на шаги, которые выполняются в некоторой последовательности.
- Используя процедурный язык, программист определяет языковые конструкции для выполнения последовательности алгоритмических шагов.

Структурное программирование

- Любая программа представляет собой структуру, построенную из трёх типов базовых конструкций: **последовательное исполнение**, **ветвление** и **цикл**.
- Базовые конструкции могут быть **вложены** друг в друга.
- Повторяющиеся или логически целостные фрагменты программы могут оформляться в виде **подпрограмм (модулей)**. В этом случае в тексте основной программы, вместо помещённого в подпрограмму фрагмента, вставляется **инструкция вызова подпрограммы**.
- Разработка программы ведётся пошагово, методом **«сверху вниз»**.

Модульное программирование

- Разбиение сложной задачи на некоторое число более простых подзадач и составлении программ для их решения независимо друг от друга.
- **Модульность** является одним из основных принципов построения программных проектов.
- **Модуль** - отдельная функционально законченная программная единица, некоторым образом идентифицируемая (**например по имени**) и объединяемая с другими.

Модульное программирование (2)

- Модули допускают **независимую компиляцию** (модуль должен быть откомпилирован раньше использующей его программы). Благодаря этому время компиляции большой программы использующей готовые модули, существенно сокращается. Это ускоряет их отладку (необходима многократная компиляция).
- Модуль **скрывает ("инкапсулирует")** представление и реализацию описываемых объектов. Изменения в модуле (при его настройке или адаптации к новым аппаратным возможностям) не требуют изменения пользовательских программ.

Модульное программирование (3)

- Языки программирования, поддерживающие модульный подход, описывают модуль как программную единицу, состоящую из двух основных частей - спецификации (интерфейса) и реализации.
- **Спецификация** содержит характеристики, необходимые и достаточные для использования модуля в других программах. Это позволяет использовать объекты модулей только на основе информации об их интерфейсе (не ожидая их полного описания).
- В **реализационной** части модуля описывается представление и алгоритмы обработки, связанные с теми или иными объектами модуля.

Модульное программирование (4)

- Модуль является одним из средств, облегчающих **верификацию** программ. Модуль, как средство создания абстракции, выделяет спецификацию и локализует сведения о реализации.
- Модули служат также целям создания **проблемно-ориентированного контекста** и **локализации машинной зависимости**.

Проектирование программ "сверху-вниз"

Задача: вычислить значение выражения: $A \cdot B + C \cdot B$, где A, C – матрицы размерности m x n, а B – матрица n x k.

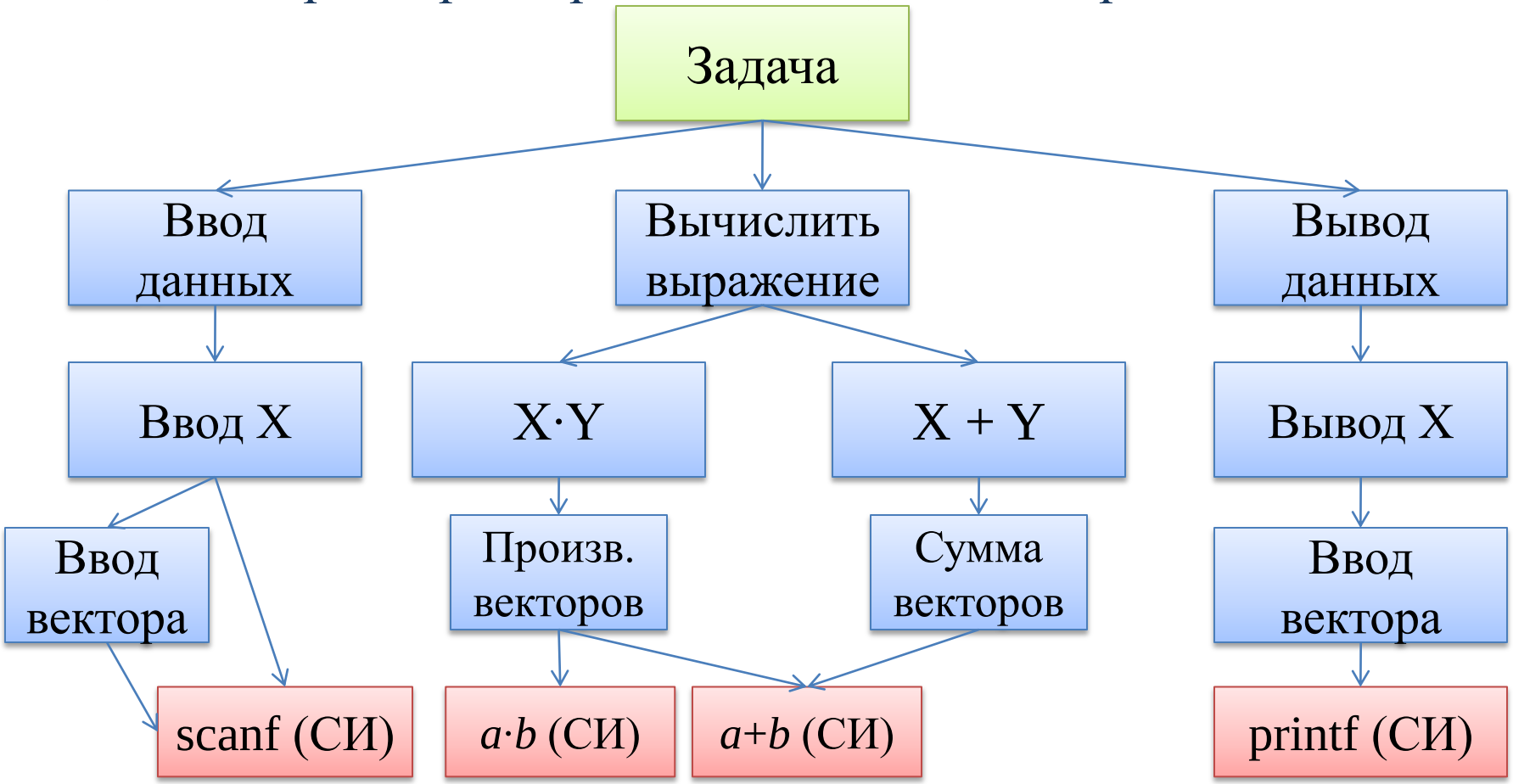
$$C = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \ddots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1k} \\ b_{21} & b_{22} & \dots & b_{2k} \\ \dots & \dots & \ddots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nk} \end{pmatrix}$$

$$c_{ij} = \sum_{l=1}^{n} a_{il} \cdot b_{lj} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + \dots + a_{nl} \cdot b_{nj}$$

c_{21}

Проектирование программ "сверху-вниз"

Задача: вычислить значение выражения: $A \cdot B + C \cdot B$, где A, C – матрицы размерности m x n, а B – матрица n x k.



Подпрограммы в языке Си

- В языке Си предусмотрены **только функции** (нет процедур). Однако с их помощью **можно** реализовать функционал процедуры.

- С функцией в языке Си связано 3 понятия:
  - **определение (реализация)** – содержит информацию необходимую для вызова функции, а также код, формирующий тело функции.
  - **объявление** (**спецификация**) – содержит **только** информацию необходимую для вызова функции (указывается в заголовочных файлах, **напр. stdio.h**).
  - **вызов** – применяется в других подпрограммах для активации действий, связанных с функцией.

РБНФ определения функции

```
ОпределениеФункции =
   ТипРезультата Имя "(" СписокФормПарам ")" "{"
   {ОператорОписания} {Оператор} "}".
ТипРезультата = ТипДанного.
Имя = Идентификатор.
СписокФормПарам =
[ТипДанного Идентиф {"," ТипДанного Идентиф }].
ТипДанного = БазовыйТип | ПользовательскийТип.
```

```
int  sum(int i,int j)
{
  return i + j;
}
```

РБНФ объявления функции (прототипа функции)

```
Прототип =
   ТипРезультата Имя "(" СписокФормПарам ")"";".
ТипРезультата = ТипДанного.
Имя = Идентификатор.
СписокФормПарам =
[ТипДанного Идентиф {"," ТипДанного Идентиф }].
ТипДанного = БазовыйТип | ПользовательскийТип.
```

```
int  sum(int i,int j);
```

РБНФ вызова функции

```
ВызовФункции =
   [Идентиф=] Имя "(" СписокФактПарам ")"";".
Идентиф = Идентификатор
Имя = Идентификатор
СписокФактПарам = [ Идентиф {"," Идентиф } ]
ТипДанного = БазовыйТип | ПользовательскийТип
```

```
int main(){
    int s, m=3;
    s = sum(m, 5);
}
```

Формальные и фактические параметры функции

Формальные параметры – **локальные переменные**, используемые внутри тела функции и **получающие значение при вызове функции** путем **копирования** в них значений соответствующих фактических параметров.

```
int  sum(int i,int j)
{
  int r = i + j;
  return r;
}
```

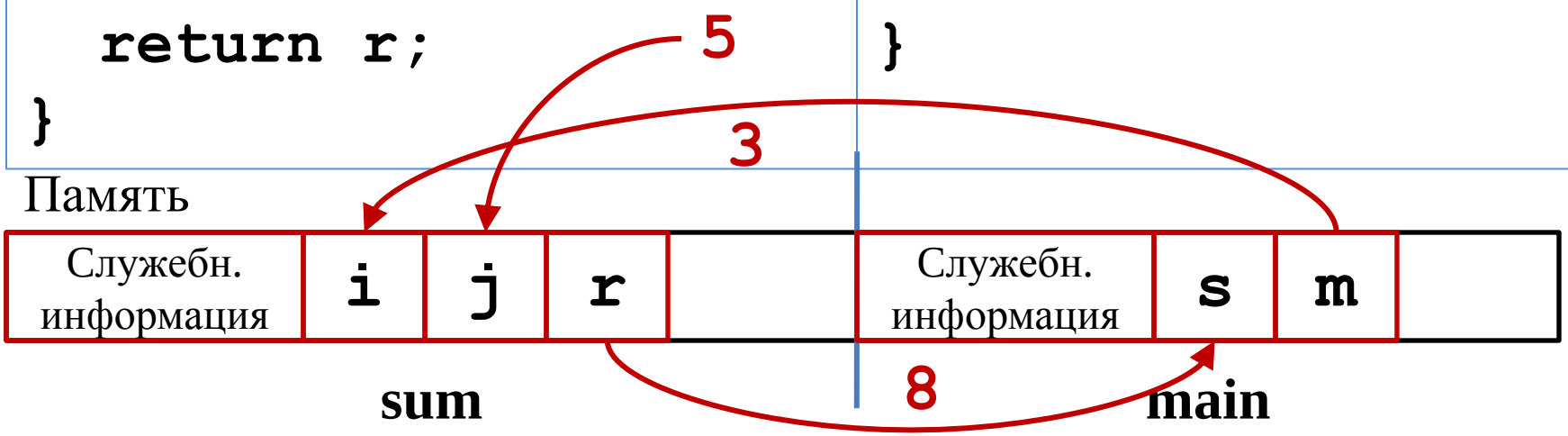
```
int main(){
   int s, m=3;
   s = sum(m, 5);
}
```

Формальные и фактические параметры функции (2)

При вызове функции значения фактических параметров **приводятся** к типу данного формального параметра.

```
int  sum(int i,int j)
{
  int r = i + j;
  return r;
}
```

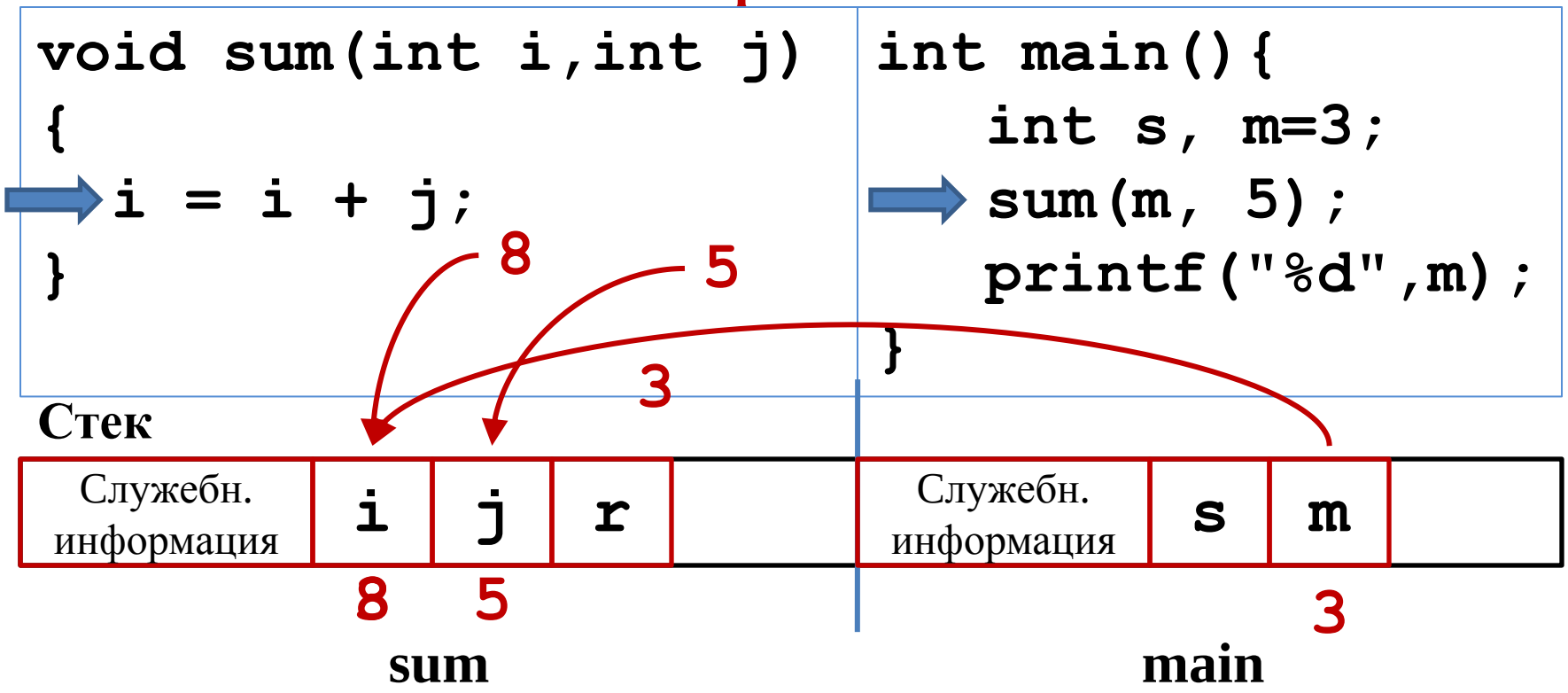
```
int main(){
   float s, m=3.8;
   s=sum(m,5.5);
}
```



Передача параметров **по значению**

В языке Си параметры передаются **по значению**. Это означает, что:

изменение формального параметра не приводит к изменению фактического!



Передача результатов через параметры функции

Если функция имеет более одного результата, возникает необходимость их возврата через параметры.

Для этого выполняется передача указателей на фактические параметры (а не их значения).

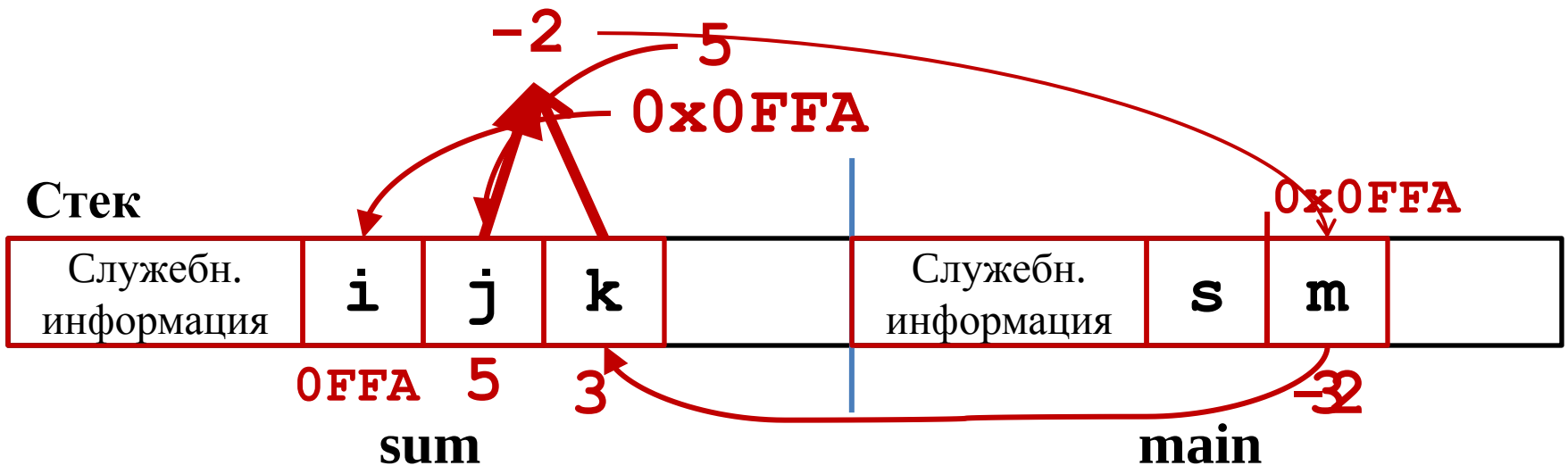
```
int sumsub(int *i,int j)
{
  int k = *i
  *i = k - j;
  return k + j;
}
```

```
int main(){
    int s, m=3;
    s = sum(&m, 5);
    printf("%d",m);
}
```

Передача результатов через параметры функции (2)

```
int sumsub(int *i,int j)
{
  int k = *i
  *i = k - j;
  return k + j;
}
```

```
int main(){
   int s, m=3;
   s = sum(&m, 5);
   printf("%d",m);
}
```



Ввод предложения с клавиатуры.

Задача:

На вход поступает английский текст *t*.

Необходимо:

Поместить в строку *s* первое предложение из текста *t*. Ввод текста осуществлять с помощью функции **scanf** (спецификатор "%s").

Пример входных данных:

This is an example!

Need to read first sentence.And skip the rest.

Sentence without end sign

Чтение входных данных

$man scanf

SCANF(3) Linux Programmer's Manual SCANF(3)

…

SYNOPSIS

#include <stdio.h>

int scanf(const char *format, ...);

. . .

Conversions

The following type modifier characters can appear in a conversion specification:

…

s Matches a sequence of non-white-space characters; ...

Чтение входных данных (2)

$man scanf

…

Conversions

The following type modifier characters can appear in a conversion specification:

…

s Matches a sequence of non-white-space characters; the next pointer must be a pointer to character array that is long enough to hold the input sequence and the terminating null character ('\0'), which is added automatically. The input string stops at white space or at the maximum field width, whichever occurs first.

Чтение входных данных (3)

$man scanf

. . .

RETURN VALUE

These functions return the number of input items successfully matched and assigned, which can be fewer than provided for, or even zero in the event of an early matching failure.

The value EOF is returned if the end of input is reached before either the first successful conversion or a matching failure occurs. EOF is also returned if a read error occurs, in which case the error indicator for the stream (see ferror(3)) is set, and errno is set indicate the error.

Чтение входных данных (выводы 1, 2)

1. Функция **scanf** при использовании спецификатора **%s** выполняет чтение последовательности символов, не содержащей пробелов (**sequence of non-white-space characters**):

```
scanf("%s", str);
```

2. С помощью поля maximum field width можно задать максимально допустимое количество читаемых символов (**The input string stops at white space or at the maximum field width, whichever occurs first**). Это важно для безопасности ввода!

```
scanf("%256s", str);
```

Чтение входных данных (вывод 3)

3. Функция scanf возвращает **количество успешно прочитанных** спецификаторов. Данная информация необходима для того, чтобы корректно считывать предложения, в которых не поставлен завершающий знак препинания:

Sentence without end sign

```
if( scanf("%256s", str) < 1 ){
   // входные данные отсутствуют
}
```

Чтение входных данных (вывод 4)

4.1. Для считывания предложения функция scanf должна быть вызвана несколько раз.

4.2. Ввод завершается, если на очередном шаге прочитана строка, содержащая завершающий знак препинания: '.', '?', '!'.

4.3. Во входных данных пробел между завершающим знаком препинания и началом следующего предложения может отсутствовать.

This is an example!

Need to read first sentence.And skip the rest.

Sentence without end sign

Пример чтения входных данных

This is an example!

1. **scanf("%256s", str) == 1 // str == "This"**
2. **scanf("%256s", str) == 1 // str == "is"**
3. **scanf("%256s", str) == 1 // str == "an"**
4. **scanf("%256s", str) == 1 // str == "example!"**
5. **scanf("%256s", str) == 0 // str == "example!"**

Пример чтения входных данных (2)

Need to read first sentence.And skip the rest.

1. **scanf("%256s", str) == 1 // str == "Need"**
2. **scanf("%256s", str) == 1// str == "to"**
3. **scanf("%256s", str) == 1 // str == "read"**
4. **scanf("%256s", str) == 1 // str == "first"**
5. **scanf("%256s", str) == 1 // str == "sentence.And"**
6. **scanf("%256s", str) == 1 // str == "skip"**
7. **scanf("%256s", str) == 1 // str == "the"**
8. **scanf("%256s", str) == 1 // str == "rest."**
9. **scanf("%256s", str) == 0 // str == "rest."**

Пример чтения входных данных (3)

Sentence without end sign

1. **scanf("%256s", str) == 1 // str == " Sentence "**
2. **scanf("%256s", str) == 1// str == "without"**
3. **scanf("%256s", str) == 1 // str == "end"**
4. **scanf("%256s", str) == 1// str == "sign"**
5. **scanf("%256s", str) == 0// str == "sign"**

Общие выводы

This is an example!

Need to read first sentence.And skip the rest.

Sentence without end sign

1. Предложение поступает на вход фрагментами, следовательно в цикле, выполняющем считывание фрагментов требуется выполнять конкатенацию (склеивание) фрагментов предложения.

2. Каждый фрагмент предложения необходимо проверять на наличие завершающего знака препинания.

3. Знак препинания может содержаться в середине фрагмента. В этом случае необходимо отбросить оставшуюся часть.

Ввод предложения

1. **Входные данные**:

- максимальное количество *n* элементов в выходной строке.

2. **Обработка данных**:

- чтение в цикле фрагментов, разделенных пробелами.
- проверка очередного фрагмента *t* на наличие признаков конца предложения, при необходимости отбрасывание лишних символов.
- конкатенация строки *s* и нового фрагмента *t*.

3. **Выходные данные**:

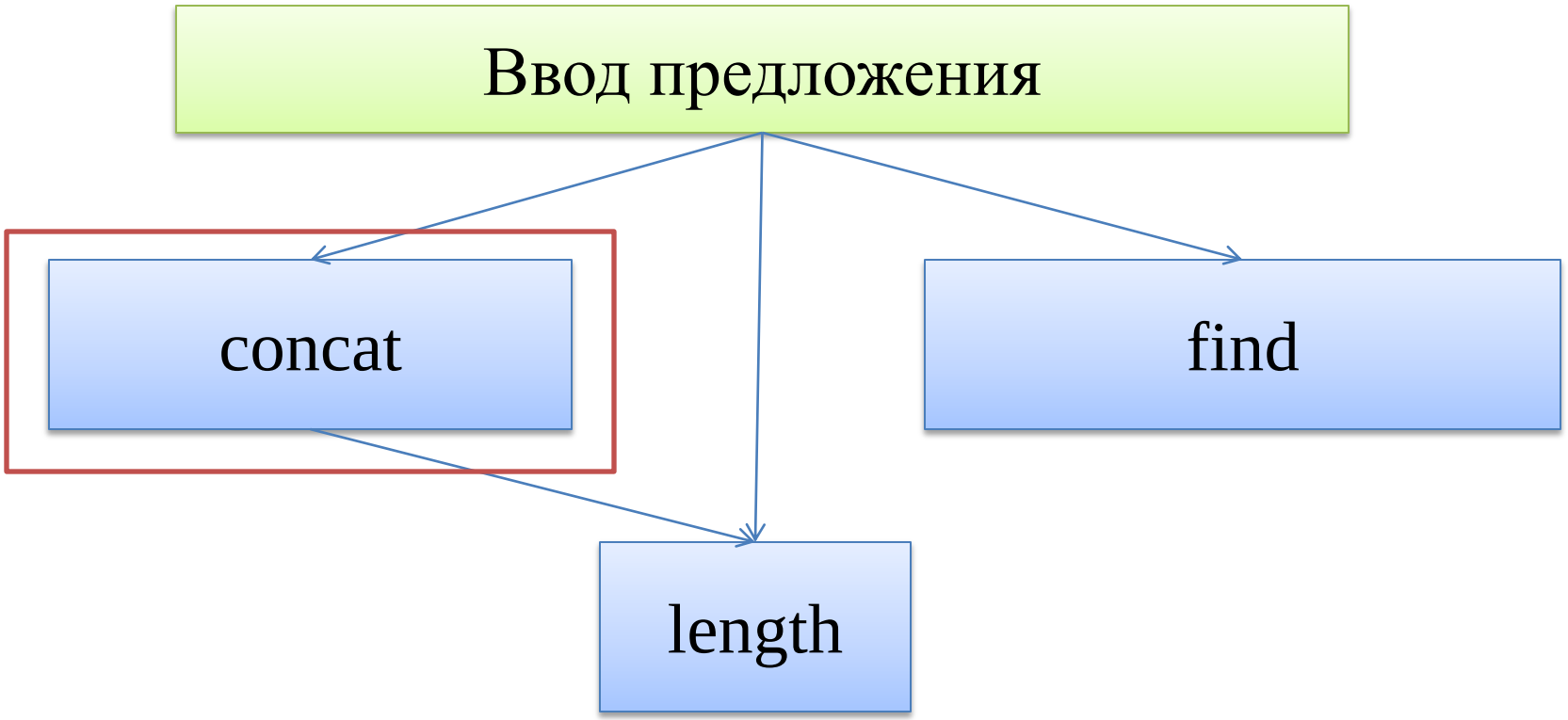
- информация об успехе/ошибке операции.
- измененная строка *d*.

Ввод предложения

sent_input(*s*, *n*)

```
1   s0 ← '\0', e ← ".?!", falg ← 1
2   do
3       if scanf("%256s",t) < 1 then flag ← 0
4       else
5           if ( i ← find(t, e, 256) ) ≥ 0 then
6               flag ← 0, t(i+1) ← '\0'
7           if concat( s, t, n) < 0 then return -1
8           if flag = 1 и length(t, 0) < 255 then concat(s, " ", n)
9   while flag
10  return length(s, n);
```

Диаграмма связей между модулями



Конкатенация строк

1. **Входные данные**:

-строка *d*, с которой производится склеивание.

-строка *s*, которая присоединяется к *d*.

- максимальное количество *n* элементов в строке *d*.

2. **Обработка данных**:

- вычисление длины *l* строки *d*.

- вычисление суффикса строки $d' = d[l,\ldots,len(s)]$.

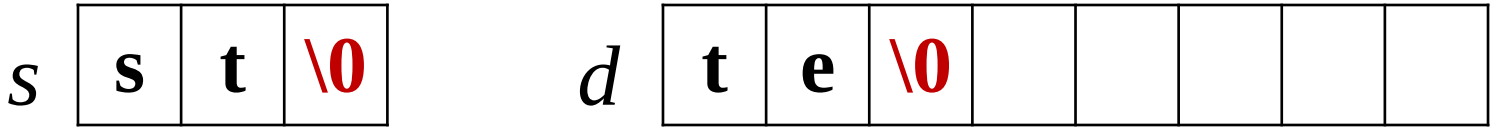
- копирование содержимого *s* в *d'*.

3. **Выходные данные**:

- информация об успехе/ошибке операции.

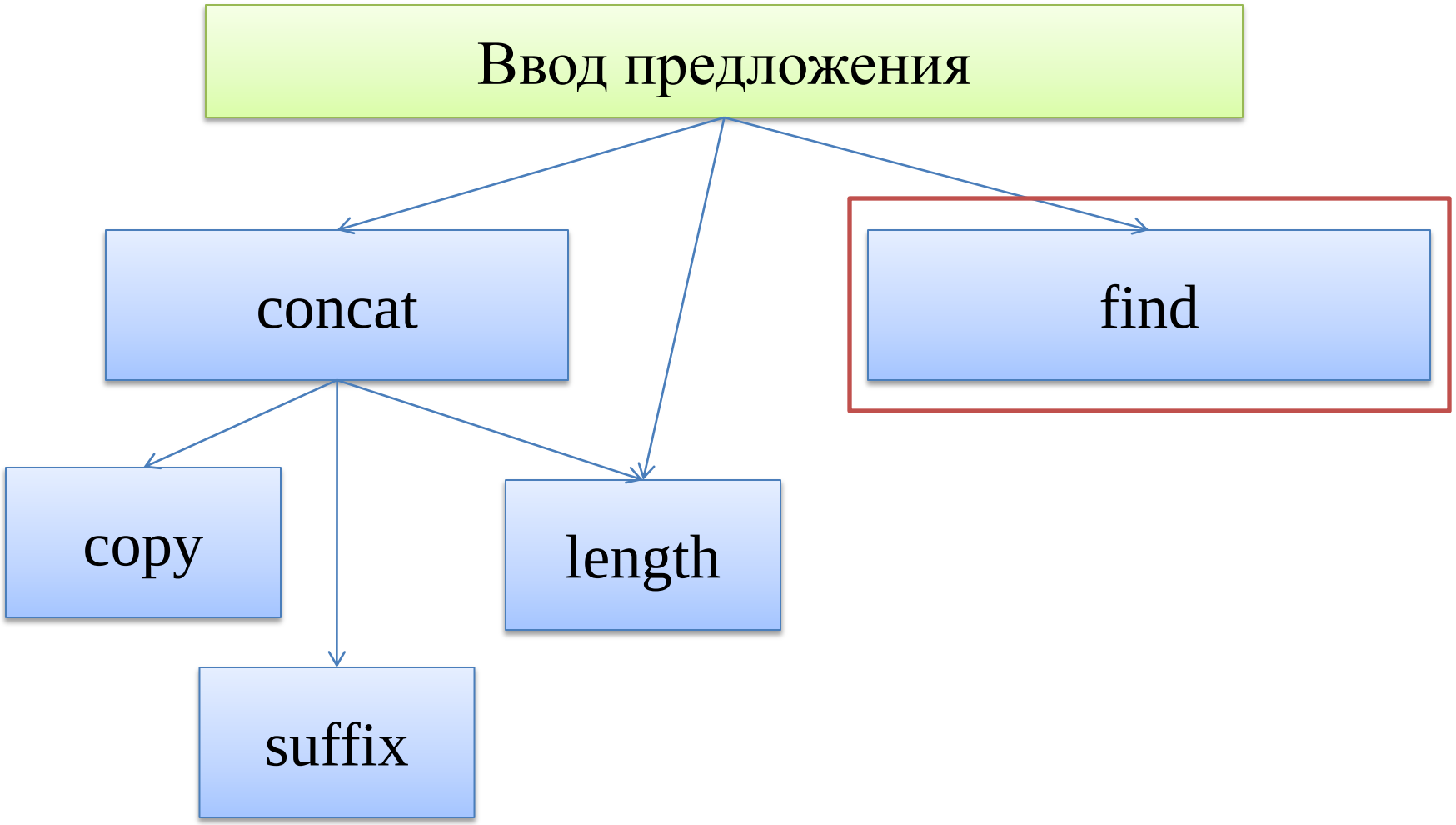
- измененная строка *d*.

Алгоритм конкатенация строк




```
concat(d, s, n)
1   i ← length(d, n)
2   if  i ≥ 0 then
3       j ← copy(suffix(d, i), s, n-i)
4   if   i < 0 или j < 0 then
5       return -1
6   else
7       return 0
```

Диаграмма связей между модулями



Поиск заданных символов в строке

1. **Входные данные**:

- просматриваемая строка s
- строка d с искомыми символами
- максимальное количество n элементов в d

2. **Обработка данных**:

-вычисление длины l_s строки s

- поиск позиции p_i символа d_i в строке s.
- выбор наименьшей позиции p_i в качестве результата.

3. **Выходные данные**:

-позиция первого вхождения символов из d в s, –1 в случае отсутствия.

Алгоритм поиска заданных символов в строке

Входные данные: просматриваемая строка *s*, строка *d* с искомыми символами, макс. длина *n* строки *s*

find(*s*, *d*, *n*)	
1	$p \leftarrow length(s, n)$
2	**for** $i \leftarrow 0; i < length(d,0) ; i \leftarrow i + 1$ **do**
3	$p \leftarrow find\_ch(s, d[i], p)$
6	**if** $p = length(s, n)$ **then**
7	**return** -1
8	**else**
9	**return** *p*

Алгоритм поиска заданных символов в строке

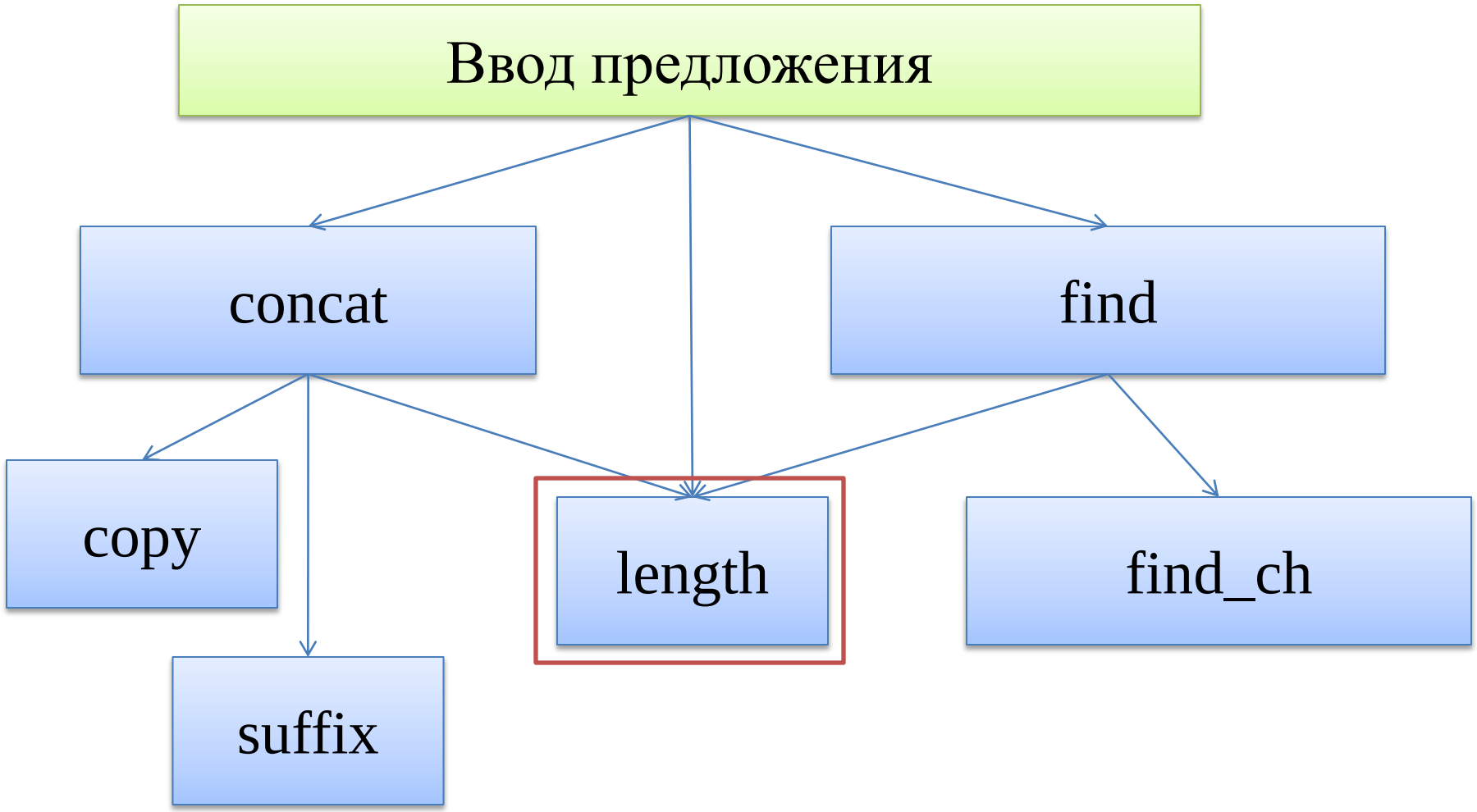
s	H	i	!	W	h	o	?	\0

*d*1	.	!	?	\0
*d*2	.	?	!	\0

find(*s*, *d*, *n*)

```
1   p ← length(s, n)
2   for i ← 0; i < length(d,0) ; i ← i + 1 do
3       p ← find_ch(s, d[i], p)
6   if p = length(s, n) then
7       return -1
8   else
9       return p
```

Диаграмма связей между модулями



Вычисление длины строки

1. **Входные данные**:

-строка s, длина которой вычисляется

- максимальное количество n элементов в строке s

2. **Обработка данных**:

- последовательный просмотр символов строки s до тех пор, пока не будет обнаружен **нуль-терминатор** или не будет просмотрено n символов.

- два варианта: если $n = 0$, то максимально допустимая длина строки не учитывается, иначе – учитывается.

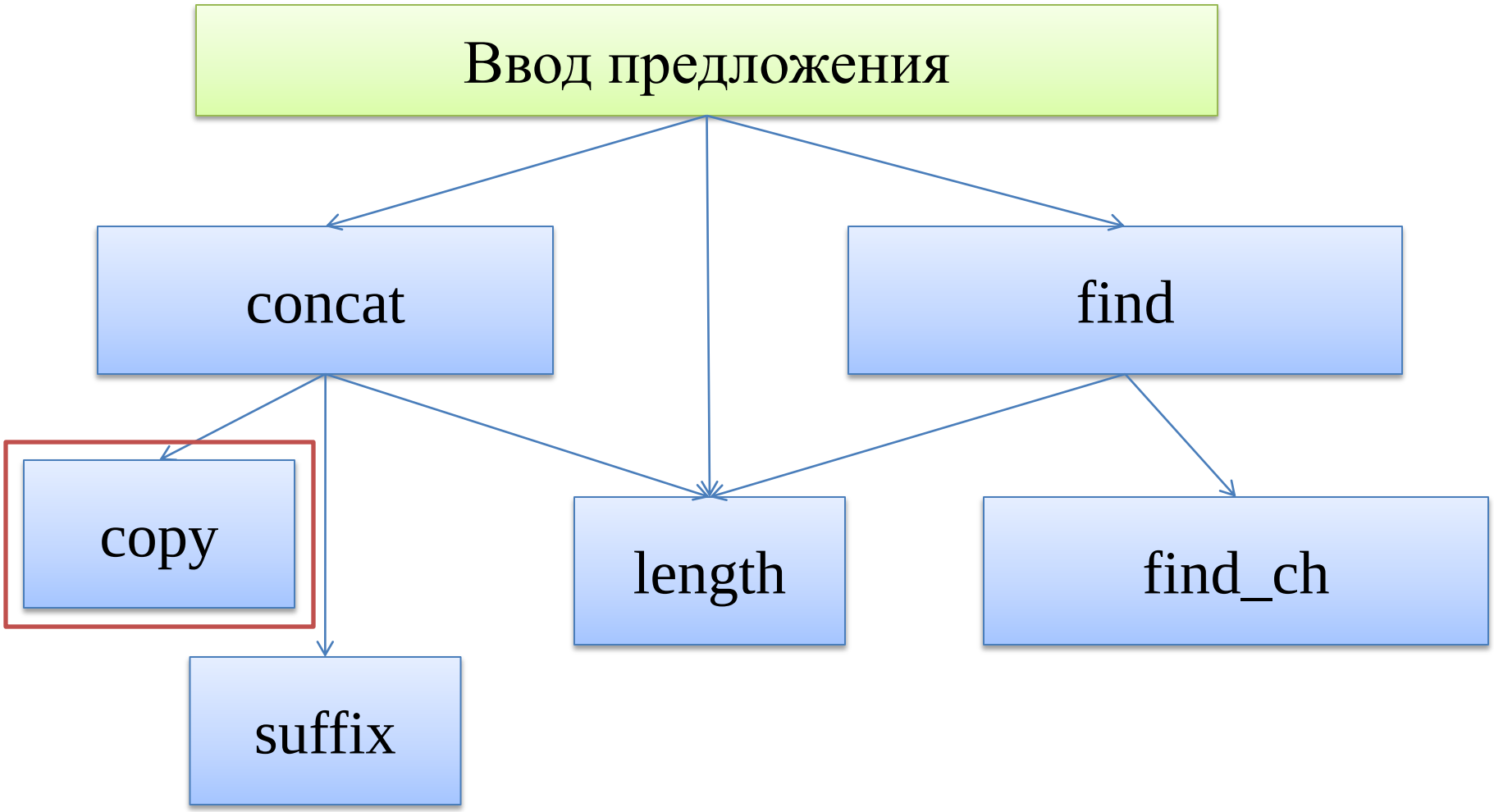
3. **Выходные данные**:

- длина строки s.

Алгоритм вычисление длины строки


```
length(s, n)
1  if n = 0 then
2      i ← 0
3      while s[ i ] ≠ '\0' do i ← i + 1
4      return i
5  else
6      i ← 0
7      while s[ i ] ≠ '\0' и i < n do i ← i + 1
8      return i
```

Диаграмма связей между модулями



Копирование строки

1. **Входные данные**:

-строка-получатель *d*.

-строка-источник *s*.

- максимальное количество *n* элементов в строке *d*.

2. **Обработка данных**:

- вычисление длины *l* строки-источника *d*.

- если l < n, то посимвольно присвоить: *d*[*i*] = *s*[*i*].

- в противном случае – вернуть -1.

3. **Выходные данные**:

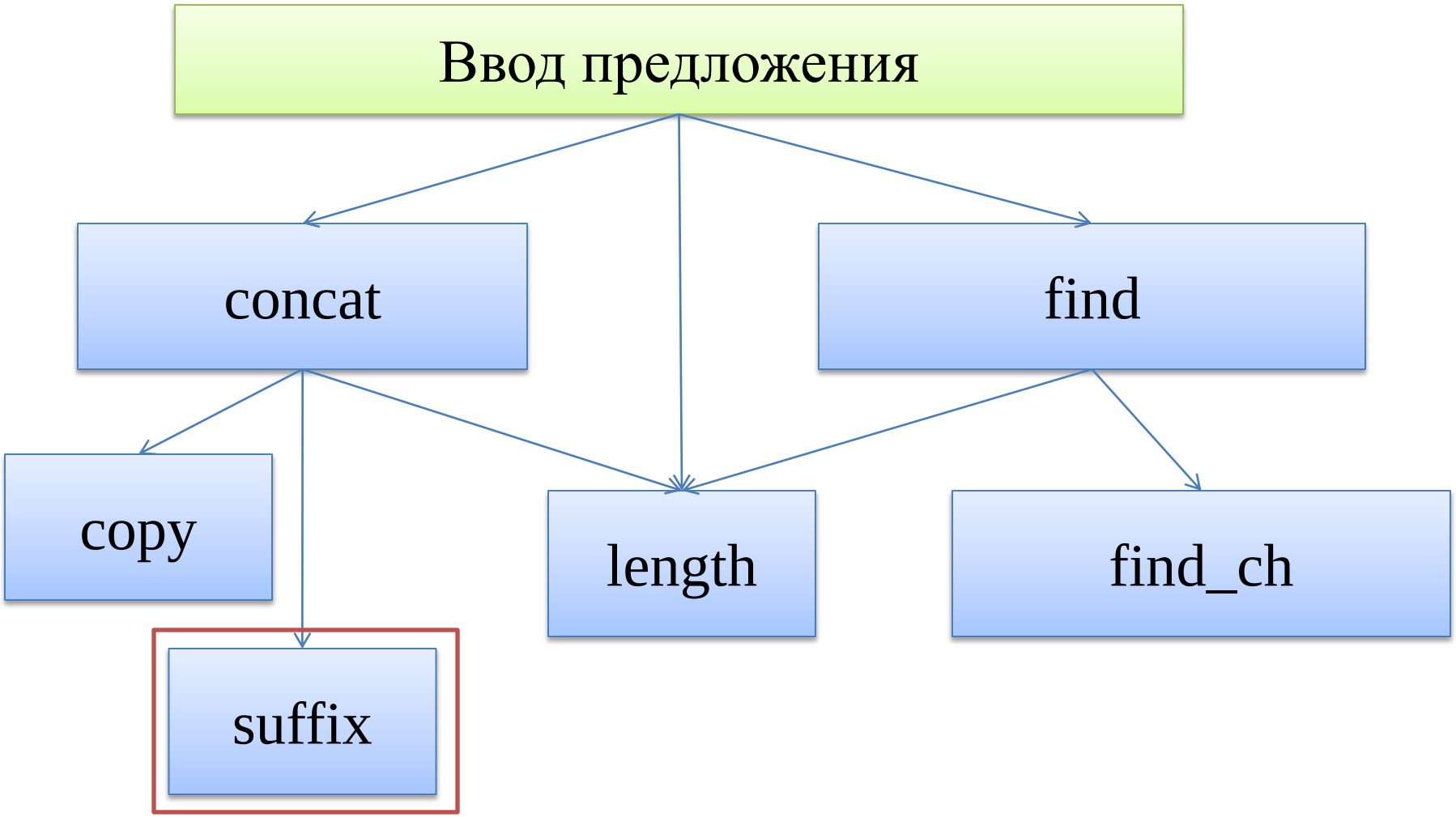
- информация об успехе/ошибке операции.

- измененная строка *d*.

Алгоритм копирования строки

find(*d*, *s*, *n*)	
1	$e \leftarrow$ *length*(*s*, 0)
2	**if** $e \geq n$ then **then**
3	**return** –1
4	**for** $j \leftarrow 0;\ j < e;\ j \leftarrow j + 1$ **do**
5	$d[\ j\ ] \leftarrow s[\ j\ ]$
6	**return** 0

Диаграмма связей между модулями



Вычисление суффикса строки

1. **Входные данные**:

- исходная строка *s*.
- индекс *i*, начала суффикса.

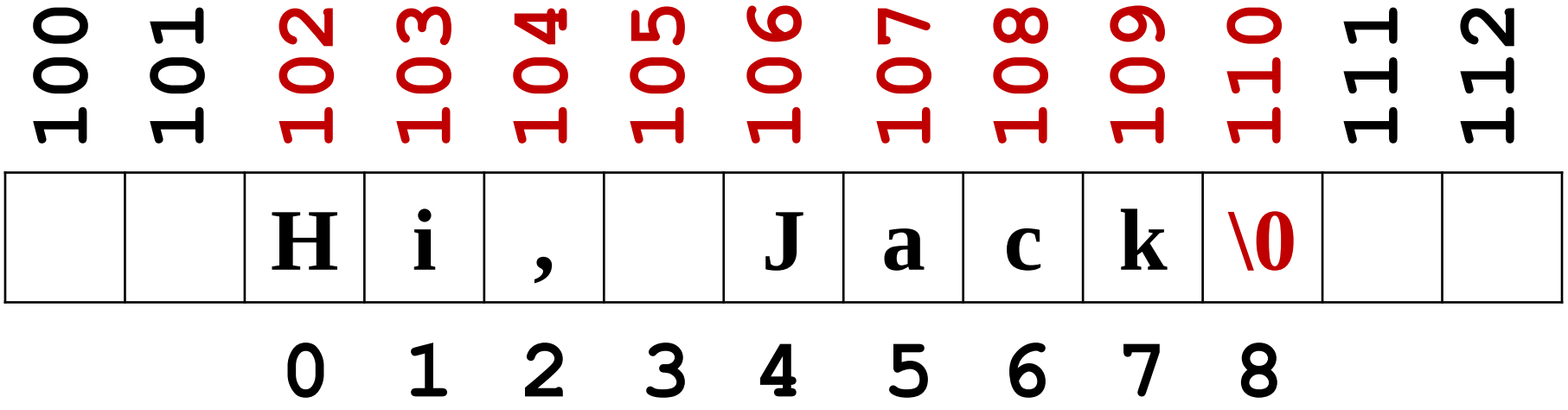
2. **Обработка данных (средствами языка Си)**:

- имя массива – указатель-константа на первый элемент массива.
- взятие адреса *i*-го элемента строки позволит рассматривать строку, которая начинается с *i*-го элемента:

&s[*i*] или (s + i)

Вычисление суффикса строки

&s[*i*] или (s + i)

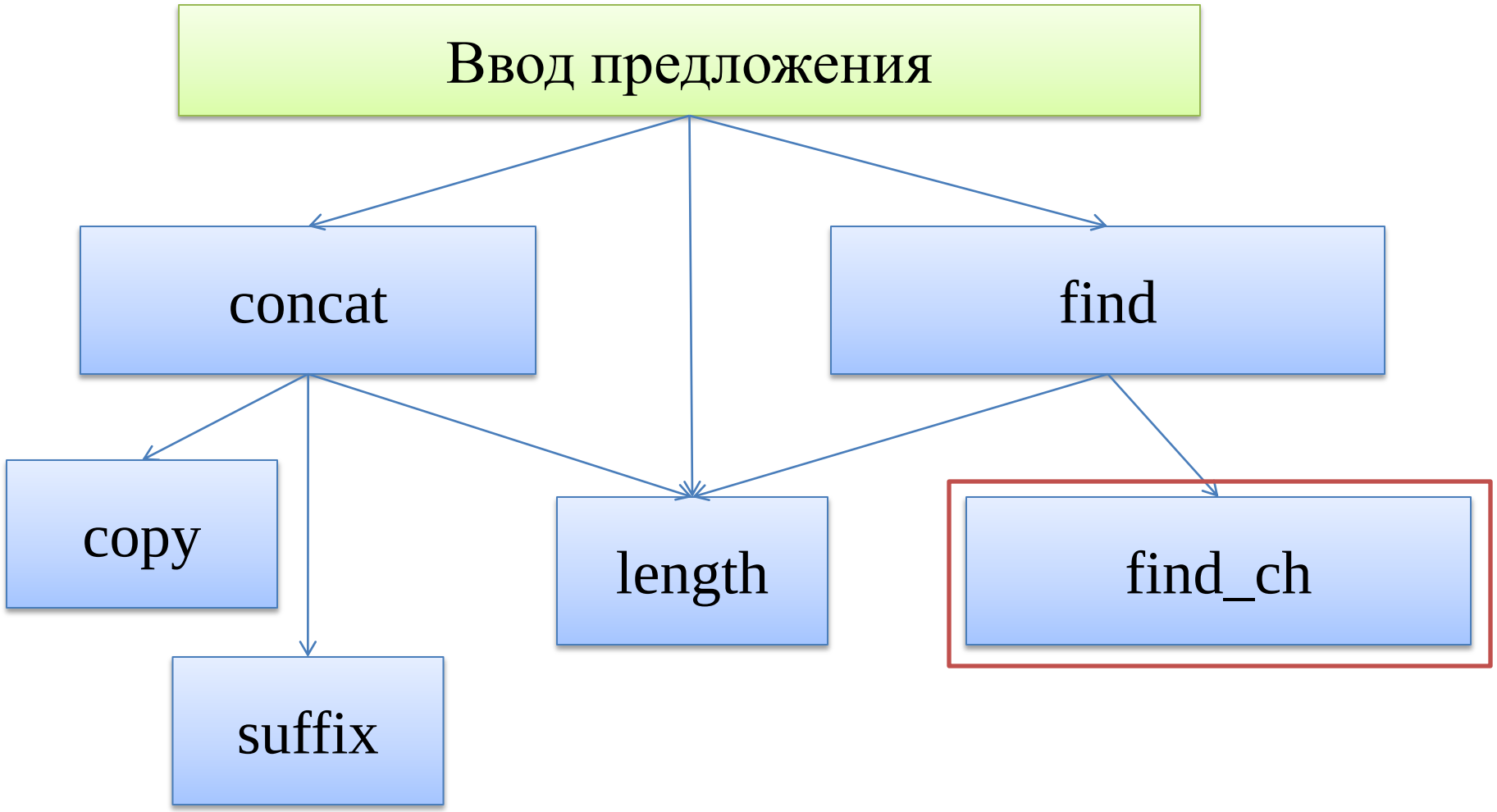


s == 102 &s[4] = 106

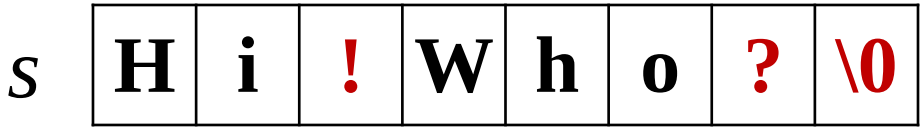
s - &s[4] = 102 – 106 = 4

&s[4] = s + 4

Диаграмма связей между модулями



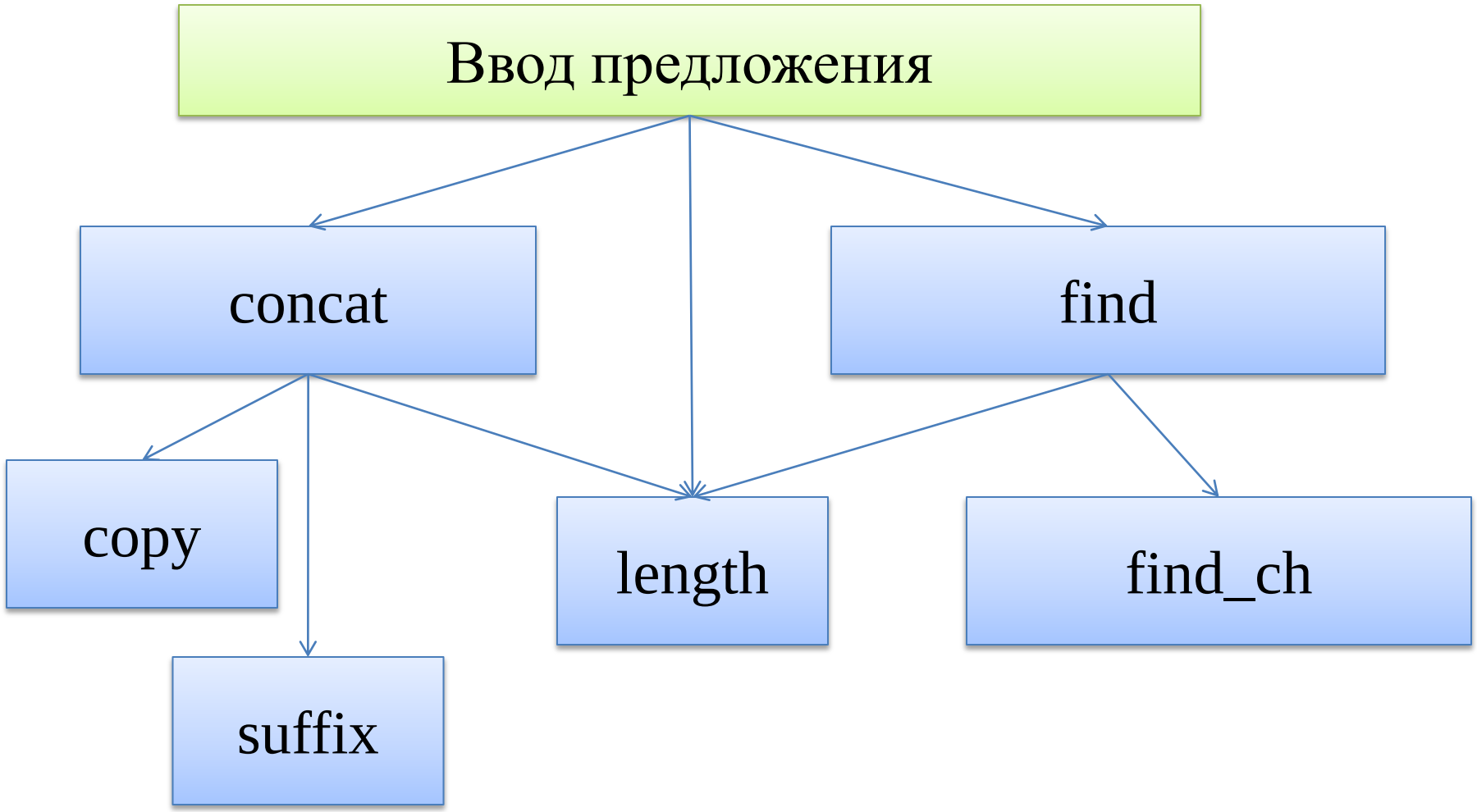
Алгоритм поиска заданного символа в строке



find_ch(s, *c*, *n*)

```
1  for i ← 0; i < n ; i ← i + 1 do
2      if s[ i ] = c then
3          n ← i
6  return n
```

Диаграмма связей между модулями



СПАСИБО ЗА ВНИМАНИЕ!