Руководство пользователя для языка Br9h

Автор документации: Люпа Ростислав

Май 2021

Содержание

1 Введение

Язык программирования `Br9h` - процедурный язык программирования высокого уровня.

Высокоуровневый язык программирования — язык программирования, разработанный для быстроты и удобства использования программистом. Основная черта высокоуровневых языков — это абстракция, то есть введение смысловых конструкций, кратко описывающих такие структуры данных и операции над ними, описания которых на машинном коде (или другом низкоуровневом языке программирования) очень длинны и сложны для понимания.

Данное руководство, по мнению автора, подойдет как начинающим программистам, желающим научиться программировать на высокоуровневом языке программирования и ознакомиться с тонкостями работы компьютера, а также тонкой работы с памятью компьютера, так и уже осведомленным в этой области людям.

В языке `Br9h` поддерживаются работа с комплексными структурами данных, работа с памятью компьютера, работа с рекуррентными вычислениями, а также есть возможность разбиения программы на отдельные файлы (модули).

Создатели языка программирования `Br9h` при разработке вдохновлялись небезызвестным языком программирования - `C++`.

2 Ключевые слова

В любом языке программирования есть зарезервированные слова, которые нельзя использовать в качестве имен переменных. И наш язык - не исключение.

Слово	**Ссылка**
if	см. п. 3.5.1
else	см. п. 3.5.1 см. п. 3.6.1 см. п. 3.6.3
elif	см. п. 3.5.1
switch	см. п. 3.5.2
for	см. п. 3.6.3
while	см. п. 3.6.1
do	см. п. 3.6.2
return	см. п. 3.4.5
break	см. п. 3.4.3
goto	см. п. 3.4.2
continue	см. п. 3.4.4
case	см. п. 3.5.2
default	см. п. 3.5.2
new	см. п. 3.1.10
delete	см. п. 3.1.10
import	см. п. 5.1
const	см. п. 3.2
void	см. п. 5.2
and	см. п. 3.1.5
or	см. п. 3.1.5
not	см. п. 3.1.5
func	см. п. 5.2
label	см. п. 3.4.1
true	см. п. 3.1.5
false	см. п. 3.1.5

Таблица 1: Список зарезервированных слов

3 Инструкции

3.1 Выражения

Выражение – это последовательность операций и их операндов, задающая вычисление.

У каждой операции есть свой приоритет, который определяет порядок выполнения операций в выражении.

Любое выражение (кроме выражений вида `delete something`) возвращает какое-либо значение.

3.1.1 Приоритеты операций

Приоритет	**Операция**	**Ассоциативность**	**Описание**
1	$	→	Унарная и бинарная операции расширения области видимости
	++	→	Постфиксный инкремент
	−−	→	Постфиксный декремент
	()	→	Вызов функции
	[]	→	Операция индексации в массиве
	.	→	Доступ к элементу структуры
	cast<>()	→	Приведение к типу
2	++	←	Префиксный инкремент
	−−	←	Префиксный декремент
	∼	←	Побитовое НЕ
	!	←	Логическое НЕ
	not		
	−	←	Унарный минус
	+	←	Унарный плюс
	@	←	Разыменование указателя
	?	←	Взятие адреса
	delete	delete delete ?..	Высвобождение памяти
	new	←	Выделение памяти
3	**	→	Возведение в степень

4	/	→	Вещественное деление
	//	→	Целочисленное деление
	*	→	Умножение
	%	→	Взятие остатка
5	+	→	Сложение
	−	→	Вычитание
6	>>	→	Битовый сдвиг вправо
	<<	→	Битовый сдвиг влево
7	>	→	Больше
	>=	→	Больше или равно
	<	→	Меньше
	<=	→	Меньше или равно
8	==	→	Равно
	!=	→	Не равно
9	&	→	Битовое И
10	∧	→	Битовое ИЛИ (исключающее)
	xor		
11	|	→	Битовое ИЛИ
12	&&	→	Логическое И
	and		
13	||	→	Логическое ИЛИ
	or		
14	−>	→	Логическая импликация
15	=	←	Присваивание
	**=	←	Выполнение соответствующей операции + переприсваивание переменной
	/ =		
	// =		
	* =		
	% =		
	+ =		
	− =		
	>>=		
	<<=		
	& =		
	∧ =		
	| =		
	−>=		

Таблица 2: Таблица приоритетов операций

3.1.2 Первичные выражения

Операнды любого оператора могут быть другими выражениями или первичными выражениями (например, в 1+2*3 операндами operator+ являются подвыражение 2*3 и первичное выражение 1).

Первичные выражения могут быть одним из следующих:

1. Литеральные константы (например 'A' или "Привет, мир");
2. Численные константы (например 12 или 0x12)

Также любое выражение в скобках также классифицируется как первичное выражение: это гарантирует, что скобки имеют более высокий приоритет, чем любой оператор. Круглые скобки сохраняют значение, тип и категорию значения.

3.1.3 Приведение типов

Существуют **явный** и **неявный** способы приведения типов. Явное приведение типов осуществляется с помощью операции `cast<type>(expr)`. Неявное приведение типов осуществляется во время операций с выражениями разных типов (например, выражение 0.3 + 5 приводится к выражению 0.3 + 5.0 и выполняется операция сложения).

Типы приводятся в порядке **возрастания приоритета** преобразования:

1. `int8`;
2. `uint8`, `char`;
3. `int16`;
4. `uint16`;
5. `int32`;
6. `uint32`;
7. `int64`;
8. `uint64`;
9. `float`;
10. `double`;
11. `bool`.

Также существуют неприводимые типы:

- любые указатели;
- строки;
- массивы.

3.1.4 Арифметические операции

Все арифметические операции вычисляют результат конкретного арифметического действия и возвращают его результат. Аргументы при этом **НЕ изменяются**.

Арифметические операции (приоритеты и значение смотри в главе 3.1.1):

- `+a`;
- `-a`;
- `a + b`;
- `a - b`;
- `a / b`;
- `a % b`;
- `a * b`;
- `a ** b`.

3.1.5 Логические операции

Логические операции вычисляют результат конкретного логического действия и возварщают его результат. Аргументы **не изменяются**.

a	**¬a**
0	1
1	0

Таблица 3: Таблица истинности логического отрицания (`not a` *или* `!a`)

a	**b**	**a ∨ b**
0	0	0
0	1	1
1	0	1
1	1	1

Таблица 4: Таблица истинности **логического ИЛИ** (`a || b` *или* `a or b`)

a	**b**	**a** $\wedge$ **b**
0	0	0
0	1	0
1	0	0
1	1	1

Таблица 5: Таблица истинности **логического И** (`a && b` *или* `a and b`)

a	**b**	**a** $\rightarrow$ **b**
0	0	1
0	1	1
1	0	0
1	1	1

Таблица 6: Таблица истинности **импликации** (`a` $\rightarrow$ `b`)

3.1.6 Операции сравнения

Возвращает логический результат сравнения значений аргументов, которые **НЕ изменяются**.

Операции сравнения (приоритеты и значение смотри в главе 3.1.1):

- `a` $==$ `b`;
- `a` $!=$ `b`;
- `a` $<$ `b`;
- `a` $<=$ `b`;
- `a` $>$ `b`;
- `a` $>=$ `b`.

Если операнды имеют арифметический тип, обычные арифметические преобразования осуществляются в соответствии с правилами для арифметических операций. Значения сравниваются после преобразования.

3.1.7 Битовые операции

Каждое число в памяти компьютера записано в виде двоичного числа. Так что в языке существуют не только арифметические операции, работающие со всем числом в целом, но и операции, выполняющиеся **побитово**.

В языке есть следующие битовые операции (приоритеты и значение смотри в главе 3.1.1):

- $\sim$`a`;
- `a` $\&$ `b`;

- `a` $\|$ `b`;
- `a` $\wedge$ `b` *или* `a xor b`;
- `a` $>>$ `b`;
- `a` $<<$ `b`.

Побитовые `НЕ`, `И` и `ИЛИ` являются, грубо говоря, операциями, которые применяют логические `НЕ`, `И` и `ИЛИ` аналогично к соответствующим разрядам чисел.

Побитовое **исключающее ИЛИ** (`XOR`) выполняет операции к соответствующим разрядам в соответствии с таблицей 7.

a	**b**	**a** $\oplus$ **b**
0	0	0
0	1	1
1	0	1
1	1	0

Таблица 7: Таблица истинности **искл. ИЛИ** (`a` $\wedge$ `b` *или* `a xor b`)

Последними побитовыми операциями, которые нужно рассмотреть, являются **операции битовых сдвигов**. Работают они с двоичным представлением числа в памяти компьютера: сдвиг `a` $<<$ `b` передвигает каждый бит числа на `b` разрядов влево, если это возможно, а последние `b` битов перезаписываются нулями; аналогично работает и сдвиг вправо, только теперь передвигаются биты вправо, а нулями перезаписываются первые `b` битов.

3.1.8 Операции присваивания

Важной частью практически любого языка программирования являются операции присваивания (или же операторы, как в `Pascal`). В нашем языке программирования **простая** операция присваивания присваивает значение переменной на значение выражения и **возвращает ссылку** на значение нашей перезаписанной переменной. Операции присваивания **с приписанной операцией** (бинарной) не просто заменяют значение переменной, а выполняют указанное действие, как будто если бы там не было знака '=' (выражение `a **= b` равносильно выражению `a = a ** b`) и перезаписывают значение переменной, возвращая указатель на нее.

Операции присваивания, присутствующие в языке, можно посмотреть под приоритетом `15` в таблице 2, глава 3.1.1.

3.1.9 Вызов функции

И, наконец, сейчас мы рассмотрим одну из самых важных операций в любом языке программирования - операция вызова функции. Любая функция (кроме процедур, то бишь функций с типом `void`) возвращает какое-либо значение. Вызов функции происходит следующим образом: пишется

имя функции, а затем в круглых скобках пишется нужное для ее вызова число аргументов (например, вызовом функции будет считаться операция `func(1, x, 13)`). При вызове функции управление программы передается этой функции, а позже и возвращается ей же назад.

3.1.10 Работа с памятью

В языке `Br9h` предусмотрена возможность работы с динамической памятью (с так называемой **кучей**). Для **выделения** памяти используется операция `new`, которая возвращает указатель на выделенную память, а для высвобождения - операция delete, которая, в свою очередь, ничего не возвращает.

К примеру возьмем выражение `ptr = new int64`. При выполнении операции выделения памяти под переменную типа `int64` в памяти в куче **начнет искаться непрерывный участок памяти** длины 8 байтов (таков размер переменной типа `int64`). Если память под переменную **не была найдена**, то **программа падает**. Затем эта память **пометится занятой** под переменную, после чего **вернется указатель** на этот участок памяти. И, наконец, адрес этого участка памяти запишется в переменную `ptr`.

Если мы потом произведем операцию `delete ptr`, то память **станет считаться свободной**.

Преимущество динамической памяти в том, что при выходе из области видимости переменной такая память автоматически не очищается. Но с динамической памятью **нужно работать аккуратно**, ведь при утере адреса участка памяти уже нельзя будет высвободить память, иначе говоря, произойдет **утечка памяти**.

3.1.11 Операции доступа

В языке предусмотрено создание пространств имен и структур, так что нужно как-то обращаться к элементам, описанным в них. В таком случае на помощь приходят **операция доступа к элементу структуры** (`str.elem`) и **операция расширения области видимости** (`nmspc$elem`, а также `$elem`).

Операция расширения области видимости может использоваться не только для доступа к именам, описанным внутри созданных пользователем пространств имен, но и в качестве **обращения к глобальному элементу** (`$global_var`), если в текущий момент в области видимости находится переменная с таким же названием.

Для работы с указателями предусмотрена операция **разыменования указателя** (`@ptr`), позволяющая получить значение элемента, на который указывает указатель.

И последней операцией является **операция индексации**, то бишь обращение к элементу массива (или строки) по его номеру (`arr[i] += 7`).

3.1.12 Декремент и инкремент

Декрементов и инкрементов бывает два типа: **постфиксные** и **префиксные**. Различаются они как приоритетом вычисления и ассоциативностью, так и поведением в общем. **Постфиксный инкремент** (декремент) возвращает **копию значения** переменной **до ее инкрементирования** (декрементирования). При последующих обращениях к этой переменной ее значение уже будет инкрементированным (декрементированным). **Префиксный инкремент** (декремент), наоборот, **сначала инкрементирует** (декрементирует) переменную, а **потом возвращает ссылку на нее**.

3.2 Оператор объявления

Объявления **вводят имена в программу** на языке `Br9h`. Объект каждого типа объявляется по-разному.

Определения - объявления, которых достаточно для использования сущности.

В языке `Br9h` существуют следующие объявления:

- объявление прототипа функции (см. главу 5.2.1);
- определение функции (см. главу 5.2.2);
- определение пространства имен (см. главу 5.7);
- определение структуры (см. главу 5.6);
- объявление переменной;

В этом разделе мы разберем объявление переменных в программе. Переменные делятся на два типа: **глобальные** (см. главу 5.4) и **локальные** (см. главу 5.5). Все переменные объявляются в соответствии с блок-схемой (см. рис. 1).

В начале можно задать переменным **спецификатор** `const`, означающий, что переменную можно инициализировать только при объявлении, то бишь последующие изменения таких переменных невозможны. Дальше задается **тип данных** переменных, затем идет **какое-либо количество символов** @, каждый из которых говорит, что переменная является указателем (проще говоря, `int8 @@@a` - указатель на указатель на указатель на переменную типа `int 8`). Затем идет **какое-либо количество пар квадратных скобок и произвольное количество запятых в этих скобках** - таким образом определяется массив и число его измерений. И, наконец, **через запятую идет описание ненулевого числа переменных**, которые **можно проинициализировать сразу** при объявлении. Имя переменной может начинаться только с буквы или нижнего подчеркивания, а дальше может состоять из того же набора символов + могут использоваться цифры. Названия переменных **не должны конфликтовать** ни с какими другими названиями.

`Кол-во пар квадратных скобок + кол-во запятых = число измерений в массиве`

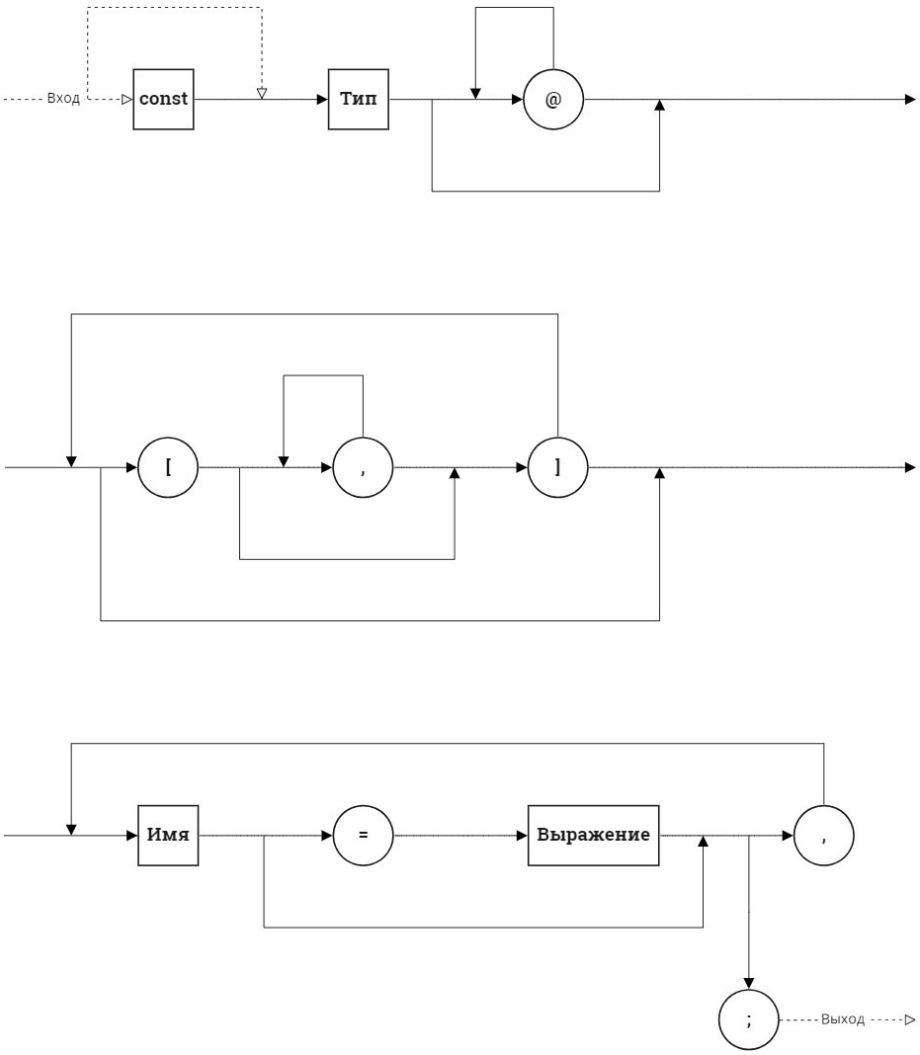

Рис. 1: Блок-схема оператора объявления переменных

3.3 Составные операторы (блоки)

Блок - очень важный оператор, использующийся в программе повсеместно: они нужны **для описания функций, структур и пространств имен**, они могут использоваться **для описания тел операторов ветвления и операторов циклов**, а также блоки могут использоваться **для обрамления части кода** в программе.

Любой блок **создает новую область видимости переменных** (см. главу 5.3).

Блок в коде программы представляет пару фигурных скобок и заключенные в них операторы:

```
{
    int8 a = 7;
}
print(a)          //
```

3.4 Операторы перехода

Операторы перехода - операторы, которые особым образом передают управление (для каждого оператора перехода поведение разное).

3.4.1 Метки

Метки в программе описываются следующим образом: |`identifier:`|. Метки могут объявляться и находиться только **внутри функций**. Метки служат точками входа для оператора перехода `goto`. Для того, чтобы поставить метку, нужно ее сначала объявить таким образом: |`label identifier;`|.

3.4.2 Оператор goto

Оператор `goto` **является универсальным оператором перехода**. Как говорилось ранее, метки создают точки входа для оператора `goto`. Это значит, что если описана и проставлена метка с именем, допустим, `name`, то можно выполнить действие |`goto name;`|, посредством чего управление передастся в место программы, в котором описана метка `name`. Нетрудно догадаться, что если метка, в которую пользователь пытается передать управление, не описана, то программа выдаст ошибку на этапе компиляции.

3.4.3 Оператор break

Следующим оператором перехода, который мы рассмотрим, будет оператор `break`. Оператор `break` используется **для принудительного выхода из циклов** (см. главу 3.6) **и из тела оператора** `switch` (см. главу 3.5.2).

В остальных случаях вызов оператора `break` будет расценен в качестве ошибки на этапе компиляции.

3.4.4 Оператор continue

Оператор `continue` применим **только в теле цикла** (см. главу 3.6) и передает управление в конец тела цикла, иначе говоря, оператор `continue` заставляет цикл закончить текущее выполнение тела цикла и **принуждает перейти к новой итерации**.

3.4.5 Оператор return

Оператор `return` применим **только внутри функций и процедур** (см. главу 5.2) и **передает управление в место, где данный экземпляр функции (процедуры) был вызван**. Для функции оператор `return` обязательно должен иметь иметь возвращаемое значение (например, |`return 0;`|), в отличие от процедур, у которых по определению не бывает возвращаемого значения (например, |`return;`|).

3.5 Операторы ветвления

3.5.1 Условный оператор if

Условный оператор if в общем случае описывается следующим образом: |`if (expr) oper`|. После описания ветки `if` может как ничего не идти, так может быть один из следующих вариантов: либо |`elif (expr) oper`|, либо |`else oper`|. Ветка `else` **всегда является конечной**.

Рассмотрим два примера:

```
if (a < b) ++a;
elif (a > b) --a;
else a = a + b;
```

```
if (a < b) {
    ++a;
} else if (a > b) {
    --a;
} else {
    a = a + b;
}
```

Эти два примера идентичны, так что далее мы не будем говорить об операторе `elif`, а будем обсуждать только операторы `else` и `if`.

Если оператор `else` после описания ветки `if` опущен, будем считать, что после ветки `if` описана ветка `else`, но представляет она из себя пустой блок.

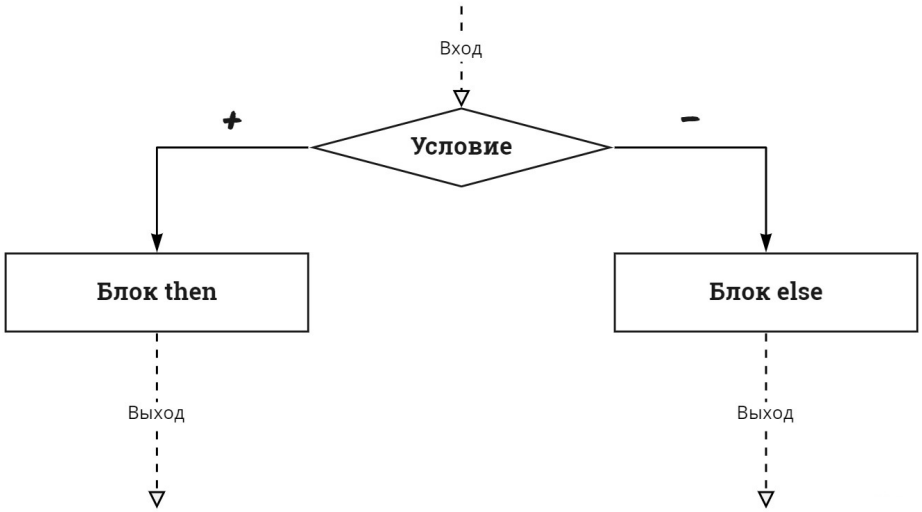

Рис. 2: Блок-схема условного оператора `if`

3.5.2 Оператор switch

Оператор `switch` в некоторых случаях является более удобным аналогом условного оператора. К примеру, если нужно вычислить значение одного конретного выражения и соотнести это значение с возможными вариантами, то оператор `switch` будет красивее и удобнее в использовании, нежели огромное множество операторов `if` и `elif`.

Для наглядности соотнесем два идентичных по функционалу участка кода:

```
switch ((a + 3) * 7) {
    case 0:
        --a;
        break;
    case 7:
        a -= 2;
        break;
    case 14:
        a -= 3;
        break;
    default:
        a = -4;
}
```

```
{
    int32 tmp = (a + 3) * 7
    if (tmp == 0) --a;
    else if (tmp == 7) a -= 2;
    else if (tmp == 14) a -= 3;
    else a = -4;
}
```

То есть оператор switch вычисляет значение выражения, которое ему подается на вход, и запоминает его на время. Дальше ищется первый описанный результат выражения, и затем выполняются все встречающиеся операторы. Оператор `break` в данном случае отвечает за выход из оператора `switch`.

То есть, если бы в первом примере после `--a` не было бы оператора `break`, то продолжилось бы выполнение операторов, которые не входят в `case 0` (иначе говоря, в данном случае выполнился бы оператор `a -= 2`).

В операторе `switch` никакой случай (`case`) не должен повторяться. Также в операторе `switch` есть аналог ветки `else`, который называется `default`. Случай `default` **может быть опущен**, как и ветка `else` в условном операторе.

3.6 Операторы циклов

Цикл - переход на более раннюю точку алгоритма. При помощи операторов перехода можно написать свой цикл, но такая реализация будет не особо читаемой. И в такой момент на помощь к нам приходят операторы циклов.

Выделяют два вида циклов:

- цикл с предусловием (цикл `while` и цикл `for`)
- цикл с постусловием (цикл do while)

3.6.1 Оператор while

Оператор `while` выполняет цикл, принцип работы которого показан на блок-схеме (см. рис. 3).

Цикл `while` описывается следующим образом: |`while (expr) oper`|. Также возможен и вариант |`while (expr) oper1 else oper2`|.

Тело цикла представляет собой *инструкцию* (блок или один оператор). При вызове оператора `break` цикл прерывается, обходя ветку `else`.

Описание ветки `else` является **необязательным** и представляет собой инструкцию, как и тело цикла.

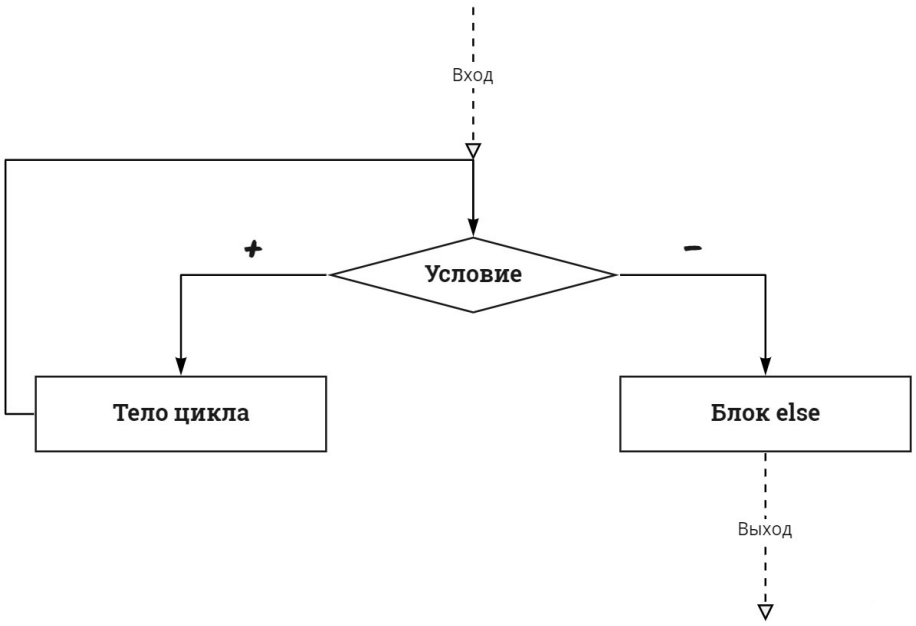

Рис. 3: Блок-схема цикла `while`

3.6.2 Оператор do while

Оператор `do while` выполняет цикл, принцип работы которого показан на блок-схеме (см. рис. 3).

Цикл `do while` описывается таким образом: |`do oper while (expr)`|.

Тело цикла представляет собой *инструкцию* (блок или один оператор). При вызове оператора `break` цикл прерывается.

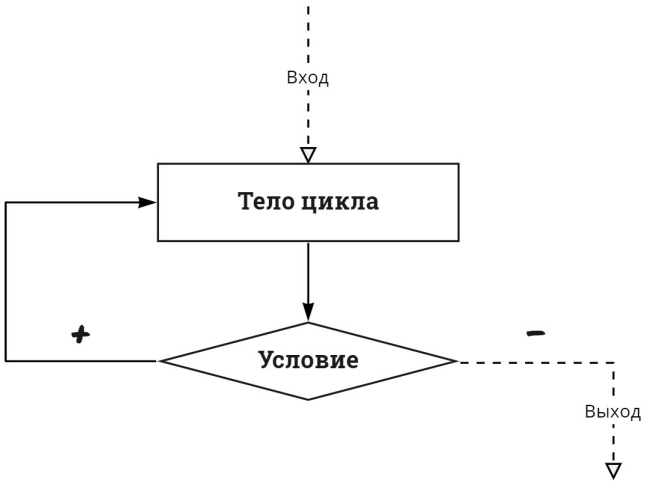

Рис. 4: Блок-схема цикла `do while`

3.6.3 Оператор for

Оператор `for` выполняет цикл, принцип работы которого показан на блок-схеме (см. рис. 5).

В первой части цикла происходит *инициализация* (возможно и создание) переменных **через запятую**. Инициализация происходит **до выполнения цикла**. Дальше записывается *условие*, при истинности которого выполняется *тело цикла*, в противном случае же цикл прерывается и выполняется *блок else* (если описан). В третьей части заголовка цикла описывается *шаговое выражение*, которое выполняется каждый раз **после** тела цикла.

Также в цикле `for` любая часть заголовка **может быть опущена**, иначе говоря, не описана.

По умолчанию (в случае, когда условие не задано) в условие записывается выражение **true**.

Тело цикла представляет собой *инструкцию* (блок или один оператор). При вызове оператора `break` цикл прерывается, обходя ветку `else`.

Описание ветки `else` является **необязательным** и представляет собой инструкцию, как и тело цикла.

Рис. 5: Блок-схема цикла `for`

Эти два примера идентичны:

```
for (init_expr; cond_expr; iteration_expr) {
    ...
}
```

```
{
    init_expr;
    while (cond_expr) {
        ...
        iteration_expr;
    }
}
```

Возможные, примерные варианты описания цикла `for`:

```
for (init_expr; cond_expr; iteration_expr) {
    ...
} else {
    ...
}
```

```
for (;;) {     # for (;;) ≡ for (; true;) ≡ while (true)
    ...
} else {       # Бесконечный цикл и при этом есть else?
    ...        # Неизвестно зачем, но если очень хочется...
}              # (если что, в ветку else невозможно зайти)
```

```
for (init_expr; cond_expr; iteration_expr) for_statement;
else {
    ...
}
```

```
for (; cond_expr; iteration_expr) {
    ...
} else else_statement;
```

```
# чаще всего используемый вариант цикла for

for (init_expr; cond_expr; iteration_expr) {
    ...
}
```

4 Типы данных

Типы данных подразделяются на два вида: **пользовательские** и **встроенные**. В языке `Br9h` создание пользовательского типа возможно только путем объявления структуры (см. главу 5.6).

В таблице 8 указаны все встроенные в язык `Br9h` типы данных и объем занимаемой памяти для каждого из типов. Под <type> @ имеется в виду указатель на любой тип данных.

Тип данных	**Размер (в байтах)**
int8	1
uint8	1
char	1
int16	2
uint16	2
int32	4
uint32	4
int64	8
uint64	8
float	4
double	8
(OSx32) <type> @	4
(OSx64) <type> @	8

Таблица 8: Таблица типов данных и их размеров

5 Структура программы

Каждая программа на языке `Br9h` устроена так, как показано на блок-схеме (см. рис. 6).

- **Препроцессор:** раздел, в котором описывается подключение дополнительных файлов;
- **Функция:** описание функции (процедуры) либо объявление прототипа функции (процедуры);
- **Глобальное описание:** объявление глобальных переменных (и их инициализация);
- **Пространство имен:** объявление и описание пространства имен;
- **Структура:** объявление и описание структуры.

Также любая программа должна иметь точку входа, то есть функцию, с которой начинается выполнение программы. В языке `Br9h` точкой входа всегда является функция |`func main() : int32;`|.

Рис. 6: Блок-схема структуры программы

5.1 Подключение дополнительных файлов

В самом начале кода программы на языке `Br9h` идет подключение дополнительных файлов (если требуется). При подключении дополнительного файла, грубо говоря, **пользователь склеивает код** того файла и и файла, к которому и идет подключение.

Примечание: `один и тот же файл не подключается несколько раз.`

Подключение файлов выглядит таким образом:

```
import "math.br9h"
import "kitties.br9h"
import "D:\\konosuba_moment\\exploooooosion.br9h"
import "C:\\Coding\\Compiler\\codegeneration.br9h"
import "C:\\Coding\\Compiler\\execution.br9h"
...
```

Если файл не удается открыть, то выдается ошибка (см. рис. 7).

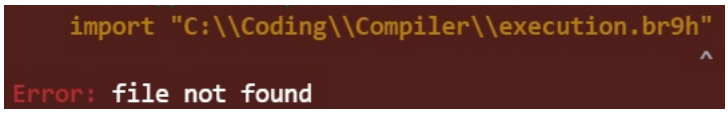

Рис. 7: Ошибка: не удалось открыть файл

5.2 Функции и процедуры

Самой важной частью любого языка программирования является наличие возможности работы с функциями и процедурами.

Функции и процедуры нужны для того, чтобы, например, описать алгоритм в общем, а потом передавать в него начальные значения переменных. То бишь **для структуризации кода, для его очищения, улучшения читаемости**.

Другим важным качеством наличия функций является возможность написания **рекурсивных алгоритмов**, то есть алгоритмов, которые вызывают сами себя, но с другими параметрами.

Для **косвенной рекурсии** (это когда две функции и более постоянно вызывают друг друга, проходя через самих себя) необходимы прототипы функций (см. главу 5.3).

5.2.1 Прототипы функций и процедур

Прототип функции - описание заголовка функции без указания ее реализации. Заголовок функции включает в себя: служебное слово `func`, название функции (идентификатор), указание формальных параметров (их может быть любое, даже нулевое, количество) в скобках через запятую, двоеточие, указание типа возвращаемого значения функции. В прототипе функции после заголовка ставится точка с запятой.

Пример прототипа функции:

```
func pow(double base, double exponent) : double;
```

Процедура отличается от функции тем, что у нее нет возвращаемого значения, и объявляется она с ключевым словом `void` вместо типа возвращаемого значения:

```
func print(char[] text) : void;
```

5.2.2 Описание функций и процедур

Описание функции (процедуры) представляет собой заголовок функции (процедуры) и составной оператор (блок), являющийся телом функции.

Описание функции в общем виде:

```
func identifier(type1 id1, type2 id2, ...) : func_type {
    ...
}
```

Пример описания функций и процедур:

- Нерекурсивная реализация алгоритма Евклида для поиска НОД(a, b) (наибольшего общего делителя чисел a и b):

```
func gcd(int64 a, int64 b) : uint64 {
    if (a == 0 || b == 0) return 0;
    if (a < 0) a = -a;
    if (b < 0) b = -b;

    int buff;
    while (b) {
        buff = b;
        b = a % b;
        a = buff;
    }
    return a;
}
```

- Простая сортировка массива по возрастанию методом пузырька:

```
func BubbleSort(int64[] array, uint32 size) : void {
    for (uint32 i = 0; i < size - 1; ++i) {
        for (uint32 j = 0; j < size - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                int64 buff = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = buff;
            }
        }
    }
    return;
}
```

5.2.3 Точка входа (main)

Точкой входа, как уже упоминалось ранее, является функция, с которой начинается выполнение программы. В языке `Br9h` такой функцией является функция с названием `main`, которая **возвращает целочисленное значение** типа `int32`. Возвращаемое значение функция имеет для того, чтобы можно было запускать нашу программу при помощи других программ и анализировать возвращаемое значение функции. По умолчанию, то есть если программа отработала корректно, функция `main` должна вернуть 0. Остальные возвращаемые значения по умолчанию считаются некорректными.

Верное описание точки входа:

```
func main() : int32 {
    ...
    return 0;
}
```

5.3 Области видимости переменных и функций

Существует два типа областей видимости переменных и функций: глобальная область видимости и локальная. Отличаются они тем, что переменные в глобальной области видимости создаются и удаляются перед выполнением основной программы и после ее выполнения соответственно, в локальной области видимости же переменные удаляются, когда исполнитель покидает эту самую область видимости.

Особенность областей видимости заключается в том, что можно использовать только переменные и функции, которые уже были объявлены в программе (то бишь были объявлены выше) и которые находятся в области видимости, в которую мы вошли, но из которой не выходили.

5.4 Объявление глобальных переменных

Глобальные переменные объявляются только в глобальной области видимости, а доступ к ним осуществляется либо просто по имени (если такое же имя не объявлено локально), либо при помощи операции расширения области видимости |`$id`|.

5.5 Объявление локальных переменных

Переменные с одинаковым именем нельзя объявлять в непосредственно одной и той же области видимости, а при обращении к переменной с именем, объявленным в разных областях видимости, выбирается та переменная, которая была объявлена позже остальных и которая еще не была удалена.

5.6 Объявление структур

Очень удобной вещью в языках программирования является возможность создания пользовательских типов данных - структур. **Структуры помогают делать код чище, проще, удобнее в использовании.**

К примеру можно взять какой-нибудь алгоритм для решения задач методом координат. Намного удобнее иметь структуру **точка**, имеющую **поля** (собственные переменные) **x**, **y** и **z**, так как появляется возможность писать функции, возвращающие сразу три элемента: тройку целых чисел, изображающую координаты точку в пространстве.

```
struct Point {
    int64 x, y, z;
};
```

Как можно заметить, структура объявляется следующим образом: сначала идет ключевое слово `struct`, затем название структуры, далее идет блок (тело структуры), в котором описаны вложенные в структуру переменные. Описание структуры всегда заканчивается точкой с запятой.

Также в структурах можно объявлять функции, принадлежащие только этой структуре, - **методы**. К примеру можно взять структуру **шар**. Опишем ее, используя предыдущую структуру **точка**:

```
struct Ball {
    struct Point {
        int64 x, y, z;
    };

    const double Pi = 3.14;
    Point central_point;
    uint64 radius;

    func Volume() : double {
        return 4 / 3 * Pi * radius ** 3;
    }
    func SurfaceArea() : double {
        return 4 * Pi * radius ** 2;
    }
} small_ball, large_ball;
```

Теперь у нашего шара, имеющего только координаты центра и модуль радиуса, можно узнать еще и объем и площадь поверхности. На примере одной такой маленькой структурки уже можно понять, что возможность для описания методов внутри структуры упрощает жизнь, делает код чище и понятней. Только поглядите:

```
small_ball.r = 3;
large_ball.r = 9;
{
    double vol1 = small_ball.Volume();
    double area1 = small_ball.SurfaceArea();
    double vol2 = large_ball.Volume();
    double area2 = large_ball.SurfaceArea();
}

{
    double vol1 = 4 / 3 * Pi * small_ball.radius ** 3;
    double area1 = 4 * Pi * small_ball.radius ** 2;
    double vol2 = 4 / 3 * Pi * large_ball.radius ** 3;
    double area2 = 4 * Pi * large_ball.radius ** 2;
}
```

Код в первом блоке намного понятнее и быстрее читается, нежели код во втором блоке.

Как уже можно было догадаться, структуры можно описывать не только в глобальной области видимости, но и внутри других структур, а также при описании структуры можно сразу же создать ее экземпляры *(примечание автора: лично я на практике так никогда не делал, но почему бы и нет?)*.

5.7 Объявление пространств имен

В языке `Br9h` также возможно объявление пользовательских глобальных (или вложенных в структуры или в другие пространства имен) областей видимости - пространств имен. Внутри них возможно объявление всего, что можно объявить в глобальном пространстве имен. Давайте опишем какое-нибудь пространство имен:

```
namespace std {
    func abs(int64 a) : uint64 {
        if (a < 0) return -a;
        return a;
    }
}
```

Для обращения к элементам пространства имен используется операция расширения области видимости:

```
print(std$abs(-7));
```

Также пространства имен можно объявлять и внутри структуры.

6 FAQ

- Кто разрабатывал язык `Br9h`?
  `Ответ:` Яицкий Г. В., Люпа Р. А., Звонков Я. С..
- Где можно скачать компилятор на язык `Br9h`?
  `Ответ:` Компилятор можно найти по этой ссылке.