Министерство образования и науки Российской Федерации
Нижегородский государственный университет им. Н.И. Лобачевского

Р.О. Митин

ЯЗЫК ПРОГРАММИРОВАНИЯ ZONNON (ОСНОВЫ)

Учебное пособие

Нижний Новгород
Издательство Нижегородского госуниверситета
2004

ББК 3973

УДК 004.432.42
М66

Рецензенты:
В.Л. Павлов, директор по аутсорсингу и поддержке процессов разработки ПО Intel Russia R&D;
В.Ю. Романов, кандидат физ.-мат. наук, научный сотрудник факультета ВМК МГУ

Митин Р.О. *Язык программирования Zonnon (основы)*. Нижний Новгород: Изд-во ННГУ им. Н.И. Лобачевского, 2004. 58 с.

ISBN 5-85746-796-9

В учебном пособии рассматриваются основные возможности языка Zonnon, необходимые для выполнения лабораторных работ по курсу «Основы алгоритмизации». Материал изложен с использованием большого количества примеров и рассчитан на непосредственную работу с языком.

Пособие предназначено для студентов младших курсов, которым предстоит решать задачи, используя язык Zonnon.

Работа выполнена в Швейцарском федеральном технологическом институте в Цюрихе (ETH) и учебно-исследовательской лаборатории «Математические и программные технологии для современных компьютерных систем (Информационные технологии)» при поддержке корпорации “Интел” и ETH.

ISBN 5-85746-796-9 ББК 3973

Введение

Целью данного пособия является краткое изложение основных возможностей языка Zonnon, необходимых для выполнения лабораторного практикума по курсу “Основы алгоритмизации”. Структура изложения предполагает непосредственную работу с примерами. Для детального знакомства со всеми возможностями языка рекомендуется обращаться к [1]. Русский перевод доступен на сайте [7].

Zonnon – язык программирования общего назначения из семейства: Pascal, Modula-2, Oberon [7]. Он, как и его предшественники, был разработан в Швейцарском федеральном технологическом институте в Цюрихе. Основной упор в нем сделан на простоту, ясный синтаксис и модульность. Унификация абстракций является основой замысла Zonnon, что отражено в его концептуальной модели, основанной на объектах, описаниях, реализациях и модулях, которые являются раздельно компилируемыми компонентами программы. Язык Zonnon добавляет новые особенности, включая процессы в объектах, перегрузку операторов и обработку исключений. Нововведения будут рассмотрены во второй части пособия “Объектно-ориентированное программирование в Zonnon”.

Автор выражает благодарность профессору Jürg Gutknecht, Е.А. Зуеву, профессору В.П. Гергелю за неоценимую помощь в разработке данного пособия. Особая благодарность Е.А. Зуеву за помощь в написании раздела “Компилятор ETH Zonnon”.

1. Алфавит языка

Алфавит – совокупность допустимых в языке символов (или групп символов, рассматриваемых как единое целое). В языке Zonnon все они формируются из множества символов стандарта ASCII. Алфавит включает в себя буквы, цифры, шестнадцатеричные цифры, специальные символы, пробелы и зарезервированные слова.

Буквы – это буквы латинского алфавита от a до z и от A до Z, а также знак подчеркивания.

Цифры – арабские цифры 0 – 9.

Шестнадцатеричная цифра имеет значение от 0 до 15. Первые 10 значений обозначаются арабскими цифрами, остальные шесть – латинскими буквами A...F или a...f.

Специальные символы – это знаки операций (+ - * / = #), разделители (, ; .), ограничители строк (" ‘). Также к специальным относятся пары символов: <= >= := и др. В программе эти пары символов нельзя разделять пробелами.

1.1. Идентификаторы

Идентификатор – имя любого объекта программы – может включать буквы, цифры и символ подчеркивания. Первый символ должен быть либо буквой, либо символом подчеркивания. В идентификаторах прописные и строчные буквы считаются различными.

В качестве идентификаторов не могут быть использованы зарезервированные слова (см. п. 1.4).

1.2. Разделители

Разделители применяются для отделения друг от друга идентификаторов, чисел, зарезервированных слов. В качестве разделителей можно использовать:

- пробел;
- любой управляющий символ (коды от 0 до 31);
- комментарий.

В любом месте программы, где можно поместить один разделитель, можно поместить любое их количество в любом сочетании. Это позволяет более наглядно оформлять программу. Рекомендуемые

приемы оформления программ на языке Zonnon читайте в рекомендациях “Zonnon Language – Coding Guidelines”, доступных на сайте [4].

1.3. Знаки пунктуации

Знак	*Примечание*
(* *)	Скобки комментария
[]	Выделение индексов массивов, элементов множеств
()	Выделение выражений, списков параметров
"	Выделение символа или строковой константы
:=	Знак присваивания значения переменной
;	Разделение предложений программы
,	Разделение элементов списка
.	Обозначение конца модуля, единицы компиляции, отделение целой части от дробной в вещественном числе, отделение полей в записи
..	Разделение границ диапазона

1.4. Зарезервированные слова

В Zonnon имеются следующие *зарезервированные слова*:

accept	**elsif**	**loop**	**refines**
activity	**end**	**mod**	**repeat**
array	**exception**	**module**	**return**
as	**exit**	**new**	**self**
await	**false**	**nil**	**send**
begin	**for**	**object**	**then**
by	**if**	**of**	**to**
case	**implementation**	**on**	**true**
const	**implements**	**operator**	**type**
definition	**import**	**or**	**until**
div	**in**	**procedure**	**var**
do	**is**	**receive**	**while**
else	**launch**	**record**	

Зарезервированные слова не могут использоваться в качестве имен идентификаторов.

2. Структура программы. Модуль

Метод программирования на Zonnon основан на четырех главных понятиях:

- **Объект** (**object**) – шаблон типа для описания объектов.
- **Описание** (**definition**) – абстрактное понятие для описания интерфейсов.
- **Реализация** (**implementation**) – контейнер для многократно используемых фрагментов реализации объекта.
- **Модуль** (**module**) – одновременно текстовый контейнер и объект структуры программы.

Эти понятия составляют базис для композиции программы в целом, а также для текстового разбиения и раздельной компиляции при разработке программы.

Текст программы включает в себя модули, объекты, описания и реализации. Исполняемая программа включает в себя один или более модулей, которые создаются динамически. Система предоставляет механизмы для динамической программной загрузки и выгрузки модулей.

Модуль (**module**) имеет двойственную природу: он определяет синтаксический контейнер для логической связи объявлений программы и одновременно описывает объект, чей жизненный цикл управляется системой (в отличие от экземпляров объектов, управляемых пользователем). Таким образом, модуль предоставляет механизм для текстового разделения исходных кодов программы и средство динамической загрузки частей программы во время работы.

Каждый модуль имеет уникальное имя и содержит текст, который может быть отдельно скомпилирован как модуль. Модуль может включать одно или более описаний, может импортировать элементы из одной или более реализаций.

Модуль может содержать:

- Описания, реализации и объекты.
- Простые объявления констант, типов, переменных, процедур.
- Описания операторов пользователя.
- Объявления активностей.

Роль “программы” в терминах языка Pascal в языке Zonnon берет на себя структура **module**.

Рассмотрим в качестве примера программу, вычисляющую сумму двух чисел:

```
module Example21;
var
  x, y, sum: integer;
begin
  write("Введите X: "); readln(x);
  write("Введите Y: "); readln(y);
  sum := x + y; (* Суммируем два числа *)
  writeln("X + Y = ", sum);
end Example21.
```

Строка

```
module Example21;
```

означает начало описания модуля с именем Example21.

Далее ключевым словом **var** начинается раздел объявления переменных

```
var
  x, y, sum: integer;
```

в котором заданы три целочисленные переменные с именами x, y, sum.

После ключевого слова **begin** находится собственно тело программы. Программа последовательно запрашивает у пользователя два числа и затем выводит их сумму. После ключевого слова **end** требуется повторить имя модуля.

3. Типы данных

Под типом данных понимается множество возможных значений и совокупность допустимых операций над ними.

В языке Zonnon можно выделить следующие группы типов:

- простые типы (целые, вещественные и др.);
- массивы (статические, динамически создаваемые, открытые);
- объекты (объекты-значения и объекты-ссылки).

Среди типов, используемых в языке, различают *стандартные* (*предопределенные*) и определяемые программистом.

К стандартным типам относятся: целые, вещественные, логические и символьные типы, множества, строки.

Все другие используемые типы данных должны быть либо объявлены в разделе объявлений типов, либо заданы непосредственно при объявлении переменных.

Раздел объявлений типов начинается зарезервированным словом **type**, после которого определяются вводимые типы. Формат задания нового типа следующий:

```
type
  <имя типа 1> = <определение типа 1>;
  <имя типа 2> = <определение типа 2>;
               . . .
  <имя типа N> = <определение типа N>;
```

Стандартные типы обозначаются предопределенными идентификаторами, которые не считаются зарезервированными словами. В языке Zonnon предусмотрены следующие стандартные типы:

- **boolean** (логический тип);
- **char** (символьный тип);
- **integer** (целые);
- **real** (вещественные числа);
- **set** (множества целых);
- **string** (строки).

Для типов **char**, **integer**, **real** и **set** число битов, используемое для хранения значения, может быть определено константным выражением в фигурных скобках **{ }** после имени типа. Для каждого типа существуют собственные предельные значения, а также значения по умолчанию (см. раздел для конкретного типа). Внутри этих типов больший тип включает значения меньшего. Например, **integer{16}** включает **integer{8}**, а **set{32}** включает **set{16}**, однако **real{80}** не включает **integer{16}**, поскольку типы несовместимы. Преобразование между различными типами должно быть явно указано (см. п. 8.6).

3.1. Целые типы

В отличие от языка Turbo Pascal, где для представления целых чисел со знаком определены пять различных типов, в Zonnon введен единственный целый тип **integer** с возможностью задания размера. Для множества неотрицательных целых в Zonnon введен тип **cardinal**. Для определения минимального и максимального

возможного значения используются встроенные функции **min()** и **max()** (см. п. 4.3.1.1). Размер по умолчанию для этих типов – 32 бита. Рассмотрим пример, выводящий границы диапазонов чисел:

```
module Example31;
type
    word = integer{16};
var i: integer;
    j: word;
begin
  i := max(integer); (* Максимальное значение *)
  writeln(i, " ", min(integer));
  j := max(word);      (* Максимальное значение *)
  writeln(j, " ", min(word));
end Example31.
```

На экран будет выведено:

```
2147483647                -2147483648
32767                     -32768
```

3.2. Логический тип

Переменная стандартного логического типа **boolean** может принимать лишь два значения, обозначаемых служебными словами **true** (истина) или **false** (ложь). При этом справедливы следующие соотношения:

```
integer(false)     =  0
integer(true)      =  1
```

Пример:

```
module Example32;
var b: boolean;
begin
  b := 2 * 2 = 4; (* Присваиваем значение
                     логического выражения *)
  writeln(" 2 * 2 = 4 это ", b);
end Example32.
```

В этом примере переменной b присваивается значение выражения «равно ли 2 * 2 четырем». На экран будет выведено:

```
2 * 2 = 4 это true
```

3.3. Множества

Элемент стандартного типа **set** в качестве своего значения может принимать любое подмножество целых чисел из интервала от **min(set)** до **max(set)**. Функции для работы с этим типом будут рассмотрены в п. 4.3.1.3.

Пример:

```
module Example33;
var  s : set ;
     s33  : set { 16 };
     s128 : set { 64 };
begin
     s := { 1 };
     s := { min(set) .. max(set) };
end Example33;
```

3.4. Символьный тип

Тип **char** позволяет работать с символами из набора, лежащего в основе реализации (платформа .NET). По умолчанию это кодировка UTF-16. На практике в .NET также используются UTF-8 и UTF-32. UTF-16 – это 16-битный тип данных **char**, применяемый для хранения Unicode-символов. Формат Unicode используется для представления символов большинства известных письменных языков. В UTF-16 коды для наиболее используемых символов располагаются в диапазоне от 0 до $FFFF_{16}$ и закодированы одним 16-битным постоянным кодом. Коды от 100000_{16} до $10FFFF_{16}$ кодируются двумя байтами, которые представляют один символ. В этом случае значение первого байта должно находиться в диапазоне от $D800_{16}$ до $DBFF_{16}$, а второго – от $DC00_{16}$ до $DBFF_{16}$. Коды в интервале от $D800_{16}$ до $DBFF_{16}$ предназначены только для этого механизма и никогда не будут использованы для обозначения символов непосредственно.

Пример:

```
module Example34;
var ch:char;
begin
    ch := 100X;
    writeln(ch);
end Example34.
```

В результате на экран будет выведен символ A.

3.5. Перечислимый тип

Перечислимый тип (или просто перечисление) представляет собой тип, возможные значения которого обозначаются идентификаторами путем явного их перечисления. При использовании в программе в качестве констант эти идентификаторы квалифицируются именем типа. Значения упорядочены, и их порядок следования определяется их последовательностью в списке перечисления. Никакое другое значение, кроме перечисленных, не относится к типу. Порядковый номер первого значения – ноль.

Пример:

```
type
     NumberKind = (Bin, Oct, Dec, Hex);
     Month = (Jan, Feb, Mar, Apr, May,
              Jun, July, Sep, Oct, Nov, Dec);
```

Использование:

```
module Example35;
type
     NumberKind = (Bin, Oct, Dec, Hex);
var
     b: NumberKind;
begin
     b := NumberKind.Bin;
end Example35;
```

Имена, заданные в различных перечислимых типах, необязательно должны быть различными, так как они всегда квалифицируются именем типа. Так, например, `NumberKind.Oct` отличается от `Month.Oct`.

Предопределенная функция *pred* возвращает значение элемента перечисления, предшествующего заданному в качестве параметра. Если значение параметра функции – первый элемент перечисления, возбуждается исключительная ситуация. Аналогично, предопределенная функция *succ* возвращает значение следующего элемента перечисления.

3.6. Строковый тип

Тип **string** в Zonnon, аналогично типу **char**, является, по существу, оберткой класса System.String из стандартной библиотеки .NET.

В .NET строка – это последовательность Unicode-символов, закодированных при помощи UTF-16, обычно используемой для представления текста. Строки в .NET строятся как последовательные коллекции, и их содержимое не может быть изменено. Результатом любой операции над строками является новая строка.

4. Выражения

Выражение – это синтаксическая единица языка, определяющая способ вычисления некоторого значения. Выражения в языке Zonnon формируются из констант, переменных, полей объектов, функций, знаков операций и круглых скобок. Сначала рассмотрим элементы, из которых строятся выражения, а затем в п. 4.5 – правила построения выражений.

4.1. Переменные

Переменными называются объекты программы, значения которых могут изменяться в процессе её выполнения. Все используемые в программе переменные должны быть объявлены с указанием их типов.

Раздел объявления переменных начинается зарезервированным словом **var**, за которым следуют объявления конкретных переменных. Каждое такое объявление состоит из имени переменной (либо из списка имен переменных), двоеточия и типа переменной (переменных). Каждое объявление завершается точкой с запятой:

```
var
     <список переменных 1>: <тип 1>;
     <список переменных 2>: <тип 2>;
             .       .       .
     <список переменных N>: <тип N>;
```

В разделе объявления можно использовать предопределенные и объявленные в программе типы. Для непредопределенных типов объявление может быть сделано как ранее в программе (в текущей или в объемлющей области действия), так и далее в текущей области действия. К предопределенным типам может быть применен модификатор ширины (см. раздел 3).

Пример объявления переменных:

```
module Example41;
type
     Figure: (Star, Rectangle, Circle, Ellipse);

var
     a, b: real;       (* Предопределенный тип *)
     x, y: integer;  (* Предопределенный тип *)
     myfigure: Figure;    (* ранее определен *)
     align: ( Top, Bottom,   (*    объявление    *)
                 Left, Right   );
begin
     align := align.Right;
     writeln( integer( align ) );
end Example41.
```

Переменные a и b имеют вещественный тип, x и y – целый, myfigure – это перечисление, тип которого задан выше, align – перечисление, заданное непосредственно в объявлении. Программа присваивает переменной перечислимого типа одно из возможных значений. На экран будет выведено число 3.

4.2. Константы

Константами называются объекты программы, значения которых не меняются в процессе её выполнения. Константы можно использовать непосредственно в виде значения или задавая идентификатор константы.

Константы могут быть целого, вещественного, символьного, логического и строкового типа. Синтаксис задания констант следующий:

```
const
     <имя константы 1> = <значение 1>;
     <имя константы 2> = <значение 2>;
             .       .       .
     <имя константы N> = <значение N>;
```

Значения констант могут задаваться как конкретными величинами, так и константными выражениями.

Примеры:

```
const
    N = 10;
    LIMIT = 2*N - 1;
    FULLSET = {min(set)..max(set)};
    COMPANY = "ETH Zurich";
    HOME = `University of Nizhni Novgorod`;
    SEPARATOR = `*`;
```

Обратите внимание на рекомендации “Zonnon Language – Coding Guidelines” [4] по использованию констант.

4.2.1. Целые константы

В изображении целых констант присутствуют лишь цифры и, возможно, знак. По умолчанию константы задаются в десятичной системе счисления. Для задания констант в шестнадцатеричной системе используется суффикс H. К константам, так же как и к переменным, может быть применен модификатор, задающий число бит, отводимых под тип данных.

Примеры целочисленных констант:

Константа	*Тип*	*Значение*
1991	**integer** / **cardinal**	1991
0DH**{**8**}**	**integer{**8**}**	13
10H	**integer** / **cardinal**	16

4.2.2. Вещественные константы

Вещественные константы всегда записываются с десятичной точкой. Допускается использование символа E в качестве показателя степени.

Примеры вещественных констант:

Константа	*Тип*	*Значение*
12.3	**real**	12.3
4.567E8	**real**	456700000
0.57712566E-6**{**64**}**	**real{**64**}**	0.00000057712566

4.2.3. Строковые и символьные константы

Строка символов – это последовательность любого, в том числе и равного нулю, количества символов из набора ASCII, расположенных на одной строке и заключенных в апострофы или кавычки. Символьные константы могут быть также определены по коду символа в шестнадцатеричной системе, завершаемому символом X. Шестнадцатеричный формат может использоваться для задания неотображаемых символов или для задания символов из расширенного набора (Unicode).

Примеры символьных констант:

```
"a", `n`, "`", `"`, 20X
```

Примеры строковых констант:

```
"Hello world",
`Language guide`,
"Don`t worry",
`"`
```

Строковые константы могут быть присвоены строковой переменной. Если внутри строковой константы необходимо использовать символ, ограничивающий её (кавычка или апостроф), то его следует удваивать. Символьные константы можно использовать везде, где допустимы строковые.

4.2.4. Константные выражения

Константные выражения – это такие выражения, которые могут быть вычислены на стадии компиляции без выполнения программы. Они являются частным случаем выражений и могут состоять из констант, знаков операций, круглых скобок и стандартных функций**.**

4.3. Стандартные функции

Мощность языка во многом определяется набором базовых объектов для конструирования программы. В языке Zonnon существует ряд заранее разработанных подпрограмм, кроме того, язык предоставляет возможность обращаться к стандартным библиотекам .NET.

4.3.1. Предопределенные функции

Предопределенные функции реализованы в стандартной библиотеке Zonnon, и их реализация в общем случае зависит от компилятора.

4.3.1.1. Арифметические функции

Имя	*Типы аргументов*	*Тип результата*	*Описание*
`abs`(x)	integer, cardinal *или* real	type of x	Модуль *x*
`dec`(v)	v: integer, cardinal *или* перечисление	Нет	v := v - 1
`dec`(v, n)	v: integer, cardinal *или* перечисление n: integer *или* cardinal	Нет	v := v - n
`inc`(v)	v: integer, cardinal *или* перечисление	Нет	v := v + 1
`inc`(v, n)	v: integer, cardinal *или* перечисление n: integer or cardinal	Нет	v := v + n
`max`(T)	integer{w}	integer	Максимальное значение integer{w}
`max`(T)	cardinal{w}	cardinal	Максимальное значение cardinal{w}
`max`(T)	Перечисление	Перечисление	Максимальное значение перечисления
`max`(T)	char{w}	integer	Максимальный символ
`max`(T)	real{w}	real	Максимальное значение real{w}

Имя	*Типы аргументов*	*Тип результата*	*Описание*
`min`(T)	integer{w}	integer	Минимальное значение integer{w}
`min`(T)	Перечисление	Перечисление	Минимальное значение перечисления
`min`(T)	char{w}	integer	Минимальный символ
`min`(T)	real{w}	real	Минимальное значение real{w}
`odd`(x)	x: integer	boolean	x **mod** 2 = 1

4.3.1.2. Строковые функции и массивы

Имя	*Типы аргументов*	*Тип результата*	*Описание*
`cap`(x)	x: char	char	Соотвествующий заглавный символ
`copy`(x, v)	x: string; v: символьный массив	Нет	v := x
`copy`(v, x)	x: string; v: символьный массив	Нет	x := v
`len`(v, n)	v: array; n: integer *или* cardinal const	integer	Количество элементов в измерении n (первое измерение = 0)
`len`(v)	v: array	integer	Аналогично len(v, 0)
`low`(x)	x: char	char	Соответствующий строчный символ

Имя	*Типы аргументов*	*Тип результата*	*Описание*
size(T)	Любой тип	integer	Количество байт требуемых под переменную типа T

4.3.1.3. Функции над множествами

Имя	*Типы аргументов*	*Тип результата*	*Описание*
excl(v, x)	v: set; x: integer *или* cardinal type	Нет	v := v - {x}
incl(v, x)	v: set; x: integer *или* cardinal type	Нет	v := v + {x}
max(T)	set{w}	integer	Максимальный элемент set{w}
min(T)	set{w}	integer	0

Рассмотрим пример использования функций для работы с множествами. Пользователь вводит числа из диапазона 1..63. Ноль означает выход. При этом командой *incl* вводимые числа добавляются в множество s. Затем в цикле выводим из диапазона 1..63 все числа, присутствующие в множестве:

```
module Example331;
var     s: set{64};
        n: integer;
begin s := {}; (* Пустое множество *)
    writeln(" Вводите числа < 64. 0 - выход.");
    repeat write("> "); readln(n); incl(s,n);
    until n = 0;
    writeln;
    for n:= 1 to 63 do
        if n in s then write(n:3); end;
    end;
    writeln;
end Example331.
```


4.3.1.4. Функции для величин порядкового типа

Имя	*Типы аргументов*	*Тип результата*	*Описание*
pred(x)	x: integer	integer	x – 1
pred(x)	x: перечисление	type of x	Предыдущий элемент перечисления
pred(x)	x: char	char	Предыдущий символ
succ(x)	x: integer *или* cardinal	integer	x + 1
succ(x)	x: перечисление	type of x	Следующий элемент перечисления
succ(x)	x: char	char	Следующий символ

4.3.1.5. Отладочные функции и функции управления исполнением

Имя	*Типы аргументов*	*Тип результата*	*Описание*
assert(b)	b: boolean	Нет	Если не b, то останов
assert(b, n)	b: boolean; n: integer *или* cardinal	Нет	Если не b, то останов с сообщением n окружению
halt(n)	n: integer *или* cardinal const	Нет	Останов программы с кодом завершения n

В *assert*(x, n) и *halt*(n) интерпретация *n* может зависеть от реализации.

4.3.1.6. Ссылки на объекты

Имя	*Типы аргументов*	*Тип результата*	*Описание*
box(v)	v: value object	type of v	Преобразует ссылку в запись
unbox(v)	v: ref object	type of v	Преобразует запись в ссылку

4.3.2. Функции стандартной библиотеки

Язык Zonnon предоставляет возможности доступа к средствам базовой для конкретной реализации операционной среды. Первый Zonnon-компилятор реализован для платформы .NET и тем самым обеспечивает доступ к ее стандартной библиотеке, которая содержит большое количество полезных классов и функций.

Для обращения к стандартным средствам Zonnon-модуль должен явно импортировать класс или пространство имен .NET (пространство имен – именованная совокупность логически связанных программных ресурсов).

Пример:

```
module Example432;
import System;  (* System – имя пространства имен,
 объединяющего все доступные ресурсы .NET *)
begin
   writeln(System.Math.Abs(-10));
end Example432.
```

Используя возможность задания псевдонимов, можно сделать эту запись более компактной (это имеет смысл в случае доступа к конкретному разделу стандартной библиотеки). Пример:

```
module Example432b;
import System.Math as Math;
begin
   writeln(Math.Abs(-10));
end Example432b.
```

Пространство имен *System* содержит много полезных классов. Одним из них является класс *Console*, который используется при создании консольных приложений. Функции *read* и *write* языка

Zonnon на самом деле являются удобной оберткой для работы с этим классом.

4.4. Операции

Все операции в Zonnon можно разбить на две группы: предопределенные и определяемые пользователем. Совместное использование предопределенных и пользовательских операций в данной части пособия рассматриваться не будет. За дополнительной информацией можно обратиться к “Zonnon Language Report” [1].

4.4.1. Арифметические операции

Операции **+**, **–**, ***** и **/** применяются к операндам численных типов. Результат имеет тип того операнда, который включает в себя тип другого операнда. Исключение составляет операция деления, результат которой имеет тип наименьшего действительного типа, который включает в себя типы обоих операндов.

Арифметические операции применимы только к величинам целых и вещественных типов. Их можно разделить на унарные и бинарные.

Унарный знак “плюс” (**+**), поставленный перед величиной целого или вещественного типа, не оказывает на неё никакого влияния.

Унарный знак “минус” (**–**) приводит к изменению знака величины.

К целочисленным и вещественным операндам применимы следующие операции:

+	сумма;
–	разность;
*****	произведение;
/	вещественное частное.

Операции **div** и **mod** применимы только к целочисленным операндам:

div	целое частное;
mod	остаток от деления целых чисел.

Математически верно, что `a = (a div b)*b + a mod b`.

Операции **+**, **–**, ***** используются также и с другими типами операндов, но тогда они имеют иной смысл (например, для строк, см.

п. 4.4.4). Эти операции также могут быть переопределены пользователем.

4.4.2. Логические операции

Эти операции применяются к операндам типа **boolean** и дают результат этого же типа.

or логическая дизъюнкция (логическое “или”)
p or q означает “если p, то **true**, иначе q”

& логическая конъюнкция (логическое “и”)
p & q означает “если p то q, иначе **false**”

~ отрицание **~** p “не p” (логическое “не”)

4.4.3. Операции над множествами

Операции над множествами применимы к типу **set** и дают результат этого же типа. Объявленная ширина в битах операндов **set** должна быть одинаковой. Одноместный знак “минус” обозначает дополнение x, т.е. −x обозначает множество целых между 0 и *max*(**set**), которые не являются элементами x (т.е. элементы, которые не входили в множество x, будут входить в −x, а которые входили – нет).

+	Объединение	Побитовый OR
-	Разность *(x − y = x * (−y))*	Побитовое вычитание
*****	Пересечение	Побитовый AND
/	Симметрическая разность: *x / y =* *((x - y) + (y - x))*	Побитовое “исключающее или”

Конструктор множества определяет значение множества посредством перечисления всех его элементов, если они имеют место, в фигурных скобках. Элементы должны быть целыми в области 0.. *max*(**set**). Область m..n обозначает все целые на интервале, начинающемся с элемента m и заканчивающемся на элементе n, включая m и n.

4.4.4. Операции над строками

Для работы с встроенным типом **string** в Zonnon помимо классической конкатенации (склеивания строк), обозначаемой знаком **+**, введена новая операция взятия подстроки. Операция вида **[**i..j**]** извлекает подстроку, начиная с символа в позиции i и заканчивая символом, предшествующим позиции j. Сокращенная запись **[**i**]** может использоваться для извлечения i-го по порядку символа строки. В операции взятия подстроки можно опускать один из параметров. Применение операции **[**i..**]** приведет к извлечению подстроки, начиная с позиции i до конца строки, а **[..**j**]** – с начала строки до позиции j (исключая сам j-й символ). Для компактной записи взятия длины строки введена операция **#**. Таким образом, конструкция вида **#**s означает число, равное длине строки s.

Пример:

```
module Example444;
var s,t:string; ch:char;
begin
    t := s + ".txt";
    s := t[2..5];
    s := t[0..#t];
    ch := s[1];
end Example444.
```

4.4.5. Отношения

В языке Zonnon определены следующие отношения:

= равно
неравно
< меньше
<= меньше или равно
> больше
>= больше или равно
in принадлежность множеству

Операции отношения всегда дают результат типа **boolean**.

Отношения **=**, **#**, **<**, **<=**, **>** и **>=** применимы к численным типам, **char**, (открытым) символьным массивам и строкам. Отношения **=** и **#** применимы к типам **boolean** и **set**, а также к процедурным типам

(включая значение **nil**). Конструкция вида x **in** s обозначает, что "x является элементом s", где x должен быть целого типа, а s – типа **set**.

4.5. Порядок вычисления выражений

Операции можно сгруппировать в несколько классов, с разными уровнями приоритета, которые синтаксически отличаются при использовании в выражениях. Операции с одинаковым приоритетом выполняются слева направо.

Например, x **–** y **–** z означает (x **–** y) **–** z.

Круглые скобки используются для заключения в них части выражения с целью изменения порядка выполнения операций. В выражении может быть любое количество круглых скобок.

Приоритет операторов изменяется следующим образом:

Наибольший	1. Вычисления в круглых скобках
	2. Вычисления значений функций 3. Унарные операции (**~**, унарный **–**, унарный **+**) 4. Операции умножения 5. Операции сложения
Наименьший	6. Операции отношения

Пример:

Математическое выражение $\left(\frac{(0.5b+a)\cdot(c+d)}{ax^2+bx+c}\right)\cdot\frac{1}{a}$ на языке Zonnon можно записать, например, так:

```
((0.5*b+a)*(c+d)/(a*x*x+b*x+c))/a.
```

5. Операторы языка

Операторы описывают некоторые алгоритмические действия. Тело процедуры можно представить как последовательность таких операторов.

5.1. Оператор присваивания

Присваивание заменяет текущее значение переменной новым значением, определяемым выражением. Выражение должно быть совместимо по присваиванию с переменной. Оператор присваивания записывается как **:=**.

Пример:

```
x := y;
z := a + b;
Res := (i > 0) and (i < 10);
D := Sqr(b) – 4 * a * c;
```

5.2. Структурированные операторы

5.2.1. Операторный блок

Операторный блок позволяет группировать вместе логически связанные операторы, а также вводить обработчики исключений.

```
module Example521;
var a: boolean;
begin
    read(a);
    begin (* Начало операторного блока *)
        a := ~ a;
    end; (* Окончание операторного блока *)
    write(a);
end Example521.
```

5.2.2. Условный оператор IF

Оператор **if** определяет условное выполнение некоторой последовательности операторов. Булевское выражение, предшествующее последовательности операторов, называется условием. Условие вычисляется, и если его значение равно **true**, то соответствующая последовательность операторов выполняется. Если условие не выполнено, то выполняется последовательность операторов, следующая за служебным словом **else**, если оно есть.

Общая схема условного оператора:

```
if <условие> then
    <последовательность для истинного условия>
else
    <последовательность для ложного условия>
end;
```

Сокращенная запись:

```
if <условие> then
    <последовательность для истинного условия>
end;
```

Пример:

```
module example522;
var a,b,m: integer;
begin
  write("A: "); readln(a);
  write("B: "); readln(b);
  if a > b then
    m := a
  else
    m := b
  end;
  writeln("Максимум из a и b = ", m);
end example522.
```

5.2.3. Условный оператор CASE

Оператор **case** определяет выбор и исполнение последовательности операторов. Выбор конкретной последовательности для исполнения производится на основе анализа значения (или набора значений), сопоставленного каждой последовательности. Вначале вычисляется выражение, заданное после служебного слова **case**, потом выполняется последовательность операторов, чей набор значений включает полученное значение **case**-выражения. Выражение **case** может иметь целый, символьный или перечислимый тип; наборы значений, сопоставленные операторам, должны иметь тот же тип, что и выражение. Метки **case** должны быть константами, и ни одно значение не должно встречаться более одного раза. Если значение выражения не совпадает со значением ни в одном наборе значений, выбирается последовательность операторов, следующая за символом **else**, в случае его наличия, в противном случае программа завершается аварийно. Общая схема:

```
  case <выражение> of
       <набор значений 1>: <операторы>
       <набор значений 2>: <операторы>
       <набор значений 3>: <операторы>
  else
       <операторы>
  end
```


Следующий пример выводит на экран сообщение о том, является ли введенный символ цифрой:

```
module example523;
var a: char;
begin
  write("Введите символ: "); readln(a);
  case a of
  "0".."9":
              writeln('Это цифра')
  else
              writeln('Это не цифра')
  end
end example523.
```

5.2.4. Оператор цикла WHILE

Оператор **while** определяет повторяемое исполнение последовательности операторов, пока условное выражение имеет значение **true**. Условие проверяется перед каждым исполнением последовательности операторов, и таким образом она может исполняться ноль или более раз.

Общая схема:

```
  while <условие> do
       <оператор 1>
       …
       <оператор N>
  end;
```

Следующий пример вычисляет количество цифр в натуральном числе:

```
module example524;
var num, dig: integer;
begin
  write("Введите натуральное число: ");
  readln(num);
  dig := 0;
  while num # 0 do
```

```
    inc(dig); num := num div 10
  end;
  writeln("Количество цифр: ", dig);
end example524.
```

5.2.5. Оператор цикла REPEAT

Оператор **repeat** определяет повторяемое исполнение последовательности операторов, пока не выполнится условие выхода, определяемое булевским выражением. Последовательность операторов выполняется хотя бы один раз. Общая схема:

```
repeat
      <оператор 1>
      …
      <оператор N>
until <условие>;
```

Пример защищенного ввода исходных данных в программу:

```
module example525;
var length: integer;
begin
  repeat
      write("Введите длину( > 0): ");
      readln(length);
  until length > 0;
  writeln("Принято: ", length);
end example525.
```

5.2.6. Оператор цикла FOR

Оператор **for** определяет повторяемое исполнение последовательности операторов фиксированное число раз.

При этом используется специальная переменная, называемая счетчиком цикла. Значение этой переменной изменяется в интервале заданных начального и конечного значений с заданным шагом. В случае если начальное значение больше конечного, шаг должен иметь отрицательный знак.

Синтаксис оператора **for** выглядит следующим образом:

```
for <переменная> := <начальное значение>
  to <конченое значение> by <шаг> do
  <оператор 1>
```



```
      …
      <оператор N>
    end;
```

Переменная, используемая в качестве счетчика цикла, должна быть известна в области действия, в которой находится оператор цикла.

В качестве примера рассмотрим проверку, является ли введенное число перевертышем (для этого получим перевернутое число и сравним):

```
module example526;
var m,n,a,i: integer;
begin
  write("Введите 4-значное число: ");
  readln(n);
  m := n;
  a := 0;
  for i := 1 to 4 do
    a := a * 10 + m mod 10;
    m := m div 10;
  end;
  if a = n then  writeln("Является перевертышем")
  else  writeln("Не является перевертышем")
  end;
end example526.
```

5.2.7. Оператор цикла LOOP

Оператор **loop** определяет повторяемое исполнение последовательности операторов. Оно прекращается срабатыванием оператора **exit** внутри последовательности. Оператор **exit** определяет завершение включающего его оператора **loop** и продолжение со следующего за ним (**loop**) оператора. Таким образом, оператор **exit** является контекстным, т.е. относится к тому оператору **loop,** в который вложен, хотя синтаксически с ним не ассоциирован. Оператор **loop** полезен для выражения циклов с несколькими точками выхода, где условие выхода выполняется естественным образом в середине повторяемой последовательности операторов. Общая схема:

```
loop
```

```
    <оператор 1>
    …
    <оператор k содержащий exit;>
    …
    <оператор N>
end;
```

Пример:

```
module example527;
import System;
var ch: char; h, r: real;
begin
  loop
    write("Введите высоту h:"); readln(h);
    write("Введите радиус основания r:");
    readln(r);
    writeln("Площадь равнобедренного конуса:",
                 System.Math.PI * h * r * r / 3);
    write("Завершить работу программы?(Y/N):");
    readln(ch);
    if (ch = 'Y') or (ch = 'y') then exit; end;
    writeln("****************************");
    writeln;
  end;
  writeln("Наберите q для выхода");
end example527.
```

В этом примере за значением числа π программа обращается к специальной константе пространства имен System.Math платформы .NET.

6. Структурированные типы данных

Структурированные типы данных формируются как наборы компонентов, являющихся простыми или также структурированными типами данных.

6.1. Массив

Массив представляет собой структуру, состоящую из набора элементов одного типа. Число элементов в массиве определяет его длину. Элементы обозначаются индексами, которые принимают

целочисленные значения между 0 и длиной массива минус 1. Синтаксические правила для типа массивов делают возможным конструировать три вида массивов, которые различаются спецификацией длины. Если длина принимает константное значение, то определен *статический массив*. Если длина измерения задается символом «*», то это – *динамический массив*. Наконец, если длина определяется произвольным выражением, то мы встречаемся с понятием так называемых *псевдодинамических массивов*.

По определению, все эти различные массивы являются многомерными; таким образом, элементы массива сами могут быть массивами, и смешивание спецификаций различных длин делает это в принципе доступным.

Для доступа к элементу массива используется операция **[**<индекс>**]**. В качестве индекса должно выступать целочисленное выражение со значением в интервале от 0 до значения, на единицу меньшего размера массива.

6.1.1. Статические массивы

В статическом массиве число элементов в любом измерении фиксировано и определяется константным выражением, которое задается в объявлении.

```
module example610;
const N = 100;
var i: integer;
    a: array N of integer;
begin
   for i := 0 to N - 1 do a[i] := i end;
   for i := 0 to N - 1 do write(a[i]:4) end;
end example610.
```

6.1.2. Открытые массивы

В объявлении открытого массива число элементов не задается. Этот тип используется только как тип формального параметра или тип значения, возвращаемого функцией.

Актуальный размер открытого массива может быть получен посредством предопределенной функции *len*.

Пример объявления открытого массива:

```
a: array of real;
```

Пример функции, возвращающей длину массива:

```
procedure MyLen(var arr: array of integer):integer;
begin
    return len(arr);
end MyLen;
```

6.1.3. Динамически создаваемые массивы

В динамическом массиве число элементов в любом измерении при объявлении не задается и обозначается звездочкой. Динамические массивы создаются явно; ответственность за выделение памяти для такого массива возлагается на программиста. Для создания динамического массива используется операция **new**:

```
arrayVariable := new ArrayType(len0, len1, … );
```

Подробнее об операции **new** будет рассказано в п. 7.2.

Значения длин в операции **new** должны задаваться посредством произвольных выражений целого типа. Во время исполнения вычисление каждого выражения должно давать в результате положительное целое, в противном случае будет сгенерирована ошибка времени исполнения. Длины всех измерений инициализируются нулем с целью предотвращения доступа элементов массива в массив переменных, который еще не был размещен.

В следующем примере память будет выделена, после того как пользователь введет размерность массива:

```
module example610a;
type Vector = array * of integer;
var i, n: integer;
var a: Vector;
begin
  write("Количество элементов: "); readln(n);
  (* Создание вектора, содержащего n элементов *)
  a := new Vector(n);
  (* Запрос у пользователя значений
       компонент вектора *)
  for i := 0 to len(a) - 1 do
       write("a[",i:2,"]: "); read(a[i])
  end;
  writeln;
  (* Вывод вектора на экран *)
  for i := 0 to len(a) - 1 do
       write(a[i]:3);
  end;
  writeln;
end example610a.
```

В следующем примере демонстрируется работа с массивом массивов, а также использование массивов в процедурах и функциях. Процедуры в Zonnon будут рассмотрены в разделе 9.

```
module example610b;
(* Инициализируем матрицу некоторыми числами *)
procedure InitializeMatrix(
                  var mat: array of array of real
                );
    var i, j: integer;
begin
    for i := 0 to len(mat, 0) - 1 do
        for j := 0 to len(mat, 1) - 1 do
            mat[i][j] := i * 10 + j;
        end
    end
end InitializeMatrix;
(* Выводим матрицу на экран *)
procedure PrintMatrix(
                  var mat: array of array of real
                );
    var i, j: integer;
begin
    for i := 0 to len(mat, 0) - 1 do
        for j := 0 to len(mat, 1) - 1 do
            write(mat[i][j]:3:0);
        end;
        writeln;
    end
end PrintMatrix;

var m: array 10 of array 10 of real;
begin
  InitializeMatrix(m);
  writeln;
```

```
  PrintMatrix(m);
end example610b.
```

6.2. Запись

Исторически в языках программирования семейства Pascal, Modula, Oberon запись – это тип, включающий ряд компонентов, называемых *полями*, которые могут быть разных типов. Количество полей записи может быть любым. Объявление записи выглядит следующим образом:

```
record <имя нового типа>;
     <поле записи>: <тип поля>;
     <поле записи>: <тип поля>;
end <имя нового типа>;
```

Пример:

```
record Complex;
     Re: real;
     Im: real;
end Complex;
```

К записям, как и к процедурам, применимы модификаторы области видимости. Если есть необходимость использования записи извне модуля, то требуется указать модификатор {*public*}:

```
record {public} Complex;
     Re: real;
     Im: real;
end Complex;
```

Обращение к полю записи осуществляется через символ “.”.

Пример:

```
var a: Complex;
begin
     a.Re := 10;
     a.Im := 7;
     writeln(a.Re," + ",a.Im,"i");
end.
```

7. Динамические структуры данных

Для тех случаев когда объем памяти, необходимый для хранения структуры данных, заранее неизвестен, используется динамическое выделение памяти. В языке Паскаль имеется возможность в процессе выполнения программы выделять и освобождать память для динамических структур. Платформа .NET задачу освобождения и повторного использования памяти берет на себя, освобождая программиста от необходимости явно удалять неиспользуемые динамические структуры. Этим занимается так называемый «*сборщик мусора*» (garbage collector).

7.1. Ссылки

Ссылка – переменная, которая хранит некий внутрисистемный идентификатор объекта или специальное значение **nil** как указание на отсутствие объекта. Ссылки необходимы для организации динамически создаваемых структур данных. Ссылочный тип задается модификатором *{ref}*. Пример:

Структура	*Ссылка*
`record Complex;`	`record {ref} Complex;`
`Re: real;`	`Re: real;`
`Im: real;`	`Im: real;`
`end Complex;`	`end Complex;`

Для ссылочного типа запись

```
var a: Complex;
```

не будет означать создания объекта. Переменная a будет хранить некоторый внутрисистемный идентификатор. После создания объекта операцией **new** этот идентификатор будет обозначать реальное расположение созданного объекта в памяти.

7.2. Операция NEW

Для создания объекта, заданного ссылочным типом, требуется использовать операцию **new**.

Для примера с типом Complex создание объекта будет записываться так:

```
a := new Complex;
```

Освобождать выделенную память не требуется.

7.3. Организация данных

Рассмотрим пример динамического выделения памяти.

```
module Example73;
```

```
record {ref} RefNumber;
   val: integer;
end RefNumber;

record Number;
   val: integer;
end Number;

var a,b: RefNumber;
var c,d: Number;

begin
  a := new RefNumber;
  a.val := 10;
  b := a; (* Присваивание ссылки
                  на тот же объект *)
  b.val := 11; (* изменятся как a так и b  *)
  writeln(a.val, " = ", b.val); (* 11 = 11 *)

  c.val := 15;
  d := c; (* Создается копия объекта  *)
  d.val := 16; (* Изменяется значение
                        у одной копии *)
  writeln(c.val, " # ", d.val); (* 15 # 16 *)
end Example73.
```

8. Совместимость и преобразование типов

В языке программирования Zonnon различают несколько уровней совместимости типов: эквивалентные типы, равнозначные типы, совместимость для присваивания, совместимость для некоторого оператора. Отдельно рассматривается совместимость массивов.

8.1. Эквивалентные типы

Две переменные `a` и `b` типов `Ta` и `Tb` имеют эквивалентный тип, если выполняется одно из следующих условий:

- `Ta` и `Tb` оба объявлены одним идентификатором типа;
- `Ta` объявлен эквивалентным `Tb` при декларации (`Ta = Tb`);
- идентификаторы `a` и `b` объявлены в одном и том же списке переменных, поле объекта или одном объявлении формальных параметров и не являются открытыми массивами.

8.2. Равнозначные типы

Два типа `Ta` и `Tb` считаются равнозначными, если выполняется одно из следующих условий:

- `Ta` и `Tb` имеют одинаковый тип (эквивалентный);
- `Ta` и `Tb` – открытые массивы с эквивалентными типами элементов;
- `Ta` и `Tb` – процедурные типы с совпадающими списками формальных параметров.

8.3. Совместимость для присваивания

Выражение `e`, имеющее тип `Te`, совместимо по присваиванию с переменной `v`, имеющей тип `Tv`, если выполняется одно из следующих условий:

- `Te` и `Tv` имеют одинаковый тип (эквивалентный);
- `Te` и `Tv` – числовые типы и `Tv` включает `Te`;
- `Tv` является процедурным типом, а `Te` есть **`nil`**;
- `Tv` – это процедурный тип и `Te` есть имя процедуры, список формальных параметров которой соответствует списку для `Tv`.

8.4. Совместимость массивов

Фактический параметр `a` типа `Ta` есть массив, совместимый с формальным параметром `f`, имеющим тип `Tf`, если выполняется одно из следующих условий:

- `Tf` и `Ta` имеют одинаковый тип (эквивалентный);
- `Tf` – это открытый массив, а `Ta` – любой массив, и типы их элементов совместимы.

8.5. Совместимость выражений

Для данных операторов в нижеследующей таблице указаны типы операндов, к которым они применимы. Таблица также показывает результирующий тип выражения. Тип T1 должен быть расширением типа T0.

Оператор	*Первый операнд*	*Второй операнд*	*Тип результата*
+ - *	Числовой	Числовой	Наименьший числовой тип, включающий оба операнда
/	Числовой	Числовой	Наименьший вещественный тип, включающий оба операнда
+ - * /	Множество	Множество	Множество
div mod	Целое число	Целое число	Наименьший целочисленный тип, включающий оба операнда
OR & ~	boolean	boolean	boolean
= # < <= > >=	Числовой	Числовой	boolean
	Перечисление T	Перечисление T	boolean
	char	char	boolean
= # < <= > >=	Символьный массив	Символьный массив	boolean
	Строка	строка	boolean
= #	boolean	boolean	boolean
	Множество	Множество	boolean
	Процедурный тип T	Процедурный тип T	boolean
	nil	nil	boolean
in	Целое число	Множество	boolean
implements	Объект	Описание	boolean

8.6. Явное преобразование типа

В тех случаях когда типы несовместимы по присваиванию, но нужно выполнить присваивание, используют явное преобразование типа.

Семейство типов	*Размер в битах*								
	8		16		32		64		128
decimal									M
								↗	
real					M	→	M		
				↗		↗			
integer	M	→	M	→	M	→	M		
		↗		↗		↗			
cardinal	M	→	M	→	M	→	M		
			↑	↗					
char			M						

Горизонтальными стрелочками → указаны «безопасные» преобразования, не приводящие к потере данных, которые не требуют явного указания программистом.

Все стрелочки ↗, ↑ обозначают преобразования, всегда разрешенные явным образом.

Для выполнения явного преобразования имя типа используется как встроенная функция, в качестве параметра которой передается переменная исходного типа, а тип возвращаемого значения соответствует имени функции. Необязательный второй параметр обозначает ширину преобразованного значения в битах.

Синтаксис:

```
<имя типа>(<исходное выражение>, <ширина типа>)
```

Пример:

```
integer(x + e/f, 16)
```

представит значение выражения *x + e/f* как 16-битное число типа integer. Ширина может быть опущена:

```
integer(x + e/f)
```

это значение выражения *x + e/f,* представленное 32-битным типом integer.

Целые числа не могут быть неявно преобразованы к действительным. Таким образом

```
var count, sum: integer; mean: real;
    …
mean := sum / count
```

не допускается. Требуется явно указать преобразование типа:

```
mean := real(sum) / real(count)
```

9. Подпрограммы в Zonnon

Процедура – способ задания подпрограммы в языке Zonnon. Все параметры, которые использует подпрограмма, можно подразделить на локальные и глобальные. Глобальные описаны в модуле и доступны из процедуры. Локальные описаны в самой процедуре и доступны только ей самой (они независимы для каждого вызова процедуры). Если процедура возвращает некоторое значение, то такую процедуру принято называть функцией.

9.1. Объявление процедуры

Подпрограмма-процедура предназначена для выполнения некой законченной последовательности действий.

Объявление процедуры состоит из заголовка и тела процедуры. Заголовок задает имя и список формальных параметров. Тело содержит объявления локальных идентификаторов (переменных, констант, типов) и операторы. Имя процедуры повторяется в конце описания.

```
procedure {<модификатор доступа>}
    <Имя процедуры>(<список формальных параметров);
<локальные декларации>
begin
    <тело процедуры>
end <Имя процедуры>;
```


Модификатор, следующий за ключевым словом **procedure**, определяет видимость процедуры извне модуля или объекта, в котором она описана. В языке определены следующие модификаторы:

- *private*: процедура видна только в той области, где определена (этот модификатор принят по умолчанию);
- *public*: процедура видна в той области, где определена, и может быть доступна из любой другой области при помощи оператора **import** (см. раздел 12).

В качестве примера рассмотрим процедуру вычисления максимума двух чисел:

```
module Example91;
  procedure Maximum(a,b:integer; var m:integer);
    begin
        if a > b then
            m := a
        else
            m := b
        end
    end Maximum;
var a,b,m: integer;
begin
 write("A = ");   readln(a);
 write("B = ");   readln(b);
 Maximum(a, b, m);
 writeln("max( A, B ) = ", m);
end Example91.
```

9.2. Объявление функции

Подпрограмма-функция предназначена для вычисления нового значения. В Zonnon объявление функции, в отличие от объявления процедуры, должно содержать указание типа возвращаемого значения, а тело функции обязательно должно содержать оператор **return**, возвращающий значение функции (подробнее см. п. 9.6).

Представим в виде функции процедуру из предыдущего раздела:

```
module Example92;
  procedure Maximum(a,b:integer):integer;
    begin
```

```
        if a > b then
            return a
        else
            return b
        end
    end Maximum;
var a,b:integer;
begin
 write("A = ");   readln(a);
 write("B = ");   readln(b);
 writeln("max( A, B ) = ", Maximum(a,b));
end Example92.
```

9.3. Формальные и фактические параметры

Формальные параметры подпрограммы указывают, с какими параметрами следует обращаться к данной подпрограмме: количество параметров, их последовательность и типы.

Два основных способа передачи параметра:

- по значению (подпрограмме передается копия значения аргумента, и подпрограмма сам параметр изменить не может);
- по ссылке (подпрограмме передается ссылка на объект, и процедура может его модифицировать).

9.4. Передача параметра по значению

Параметр-значение указывается в заголовке своим именем и через двоеточие – типом.

Пример функции, вычисляющей квадрат вещественного числа:

```
procedure Sqr(a: real): real;
begin
      return a * a;
end Sqr;
```

9.5. Передача параметра по ссылке

Параметр-переменная указывается в заголовке подпрограммы аналогично параметру-значению, но с добавлением перед именем параметра зарезервированного слова **var**.

Пример процедуры, обнуляющей переданный параметр:


```
procedure Null(var a: real);
begin a := 0;
end Null;
```

При передаче параметра по ссылке переданной переменной можно присваивать новые значения. Механизм передачи параметра по ссылке, как правило, используется, если нужно модифицировать передаваемую переменную, а также если передаваемая переменная имеет большой размер и нет смысла в создании её копии.

9.6. Оператор RETURN

Оператор **return** задает завершение выполнения подпрограммы и обозначается соответствующим зарезервированным словом. Для подпрограмм-функций после **return** должно следовать выражение, которое определяет значение, возвращаемое функцией. Тип выражения должен быть совместимым по присваиванию с типом результата, определенным в заголовке функции.

В процедурах оператор **return** связан с концом тела процедуры. Поэтому любой явный оператор **return** появляется как дополнительная (возможно, исключительная) точка завершения.

10. Ввод-вывод

10.1. Консоль

Язык содержит встроенные средства для простейшего текстового ввода и вывода, аналогичные предопределенным процедурам в Pascal. Это две процедуры для ввода текста – *read* и *readln* и две для вывода – *write* и *writeln*.

10.1.1. Процедуры ввода

10.1.1.1. Процедура READ

Формат процедуры *read*:

```
read (v1, …, vn)
```

Она может иметь один или более параметров, каждый из которых – это значение некоторого типа.

Если v – это символьный тип **char**, то *read*(v) считывает следующий символ из входящего текста в v.

Если v – это значение типа **integer** или **real**, то *read*(v) неявно считывает последовательность символов из входного текста и присваивает число v. Символы-разделители и перевод строки пропускаются и отбрасываются.

Для вещественных чисел разделителем во входной строке считается, знак установленный в настройках локализации вашей операционной системы (Windows, Linux). Для России это обычно запятая.

10.1.1.2. Процедура READLN

Формат процедуры *readln*:

```
readln(v1, …, vn)
```

Она обладает той же функциональностью, что и read, но после считывания vn все оставшиеся символы на строке пропускаются (включая символ перевода строки).

10.1.2. Процедуры вывода

10.1.2.1. Процедура WRITE

Формат процедуры *write*:

```
write (p1, …, pn)
```

Она может иметь один или более параметров, каждый из которых имеет формат e , e:m или e:m:n, где e представляет выводимое значение, а m и n – спецификаторы ширины поля вывода. Если значению e требуется меньше, чем m символов для своего представления, то выводятся разделители (пробелы), гарантируя, что всего будет записано m символов. Если m опущено, то будут использоваться значения по умолчанию. Формат e:m:n применим только для чисел типа **real**.

Параметры процедуры *write* могут быть типов **char**, **string**, **boolean**, **byte**, **integer** и **real**:

- Если e имеет тип **char**, то *write* (e:m) выведет m-1 пробелов, а следом – символ содержащийся в e. Если m опущено, то будет выведен только сам символ.

- Если e имеет тип **string**, то *write* (e:m) выведет строку символов предваренных пробелами для того, чтобы совокупная длина вывода была m.
- Если e имеет тип **boolean**, тогда будет выведено одно из двух слов **true** или **false** предваренное пробелами для того, чтобы совокупная длина была m.
- Если e имеет тип **integer**, то будет записано десятичное представление числа e предваренное пробелами для того, чтобы совокупная длина была m.
- Если e имеет тип **real**, то будет записано десятичное представление числа e предваренное пробелами для того, чтобы совокупная длина была m. Если параметр n пропущен, то число с плавающей точкой будет представлено в научном формате. Если n указано, то используется представление с фиксированной точкой с n цифрами после точки.

10.1.2.2. Процедура WRITELN

Формат процедуры *writeln*:

```
writeln (v1, …, vn)
```

Она имеет ту же функциональность что и *write*, но после вывода vn записывается признак конца строки.

10.1.2.3. Значения ширины по умолчанию

Ширина полей по умолчанию для процедур *write* и *writeln* зависит от типа выводимого значения. Значения по умолчанию следующие:

- **char** определяет поле шириной 1;
- **string** определяет поле шириной 4;
- **boolean** определяет поле шириной 6;
- **integer** определяет поле шириной 20;
- **real** определяет поле шириной 20.

10.2. Работа с файлами

Стандартные функции ввода-вывода в Zonnon в отличие от Pascal позволяют работать только с консолью (стандартным устройст-

вом ввода/вывода). Для работы с фалами требуется обращаться к стандартной библиотеке .NET. Пример:

```
module Example102;
import System, System.IO;
type
    SW = System.IO.StreamWriter;
var sw : SW;
    fn : System.String;
begin
    fn := "myfile.txt";
    sw := new System.IO.StreamWriter(fn, false);
    sw.Write("Text to be written to file");
    sw.Close();
    readln();
end Example102.
```

За информацией о классах пространства System рекомендуется обращаться к собранию документов компании Microsoft (MSDN) [2].

11. Пример решения задач на Zonnon

11.1. Сортировка массива

В качестве примера рассмотрим быструю сортировку.

```
module Example91;
    const MAX_SIZE = 20; (* Размер массива *)
     (*Зададим тип элемента массива и тип массив*)
    type ElementOfArray = integer;
    type DefaultArray = array MAX_SIZE
                          of ElementOfArray;
     (* Непосредственно объявление и создание
        массива*)
     var MyArray: DefaultArray;

(* Рекурсивная функция быстрой сортировки. Описание
Алгоритма смотрите в книге Н.Вирта «Алгоритмы и
структуры данных» [6] *)

     procedure QuickSort(
           var a: DefaultArray; L, R: integer);
     var i,j: integer; w,x: ElementOfArray;
```

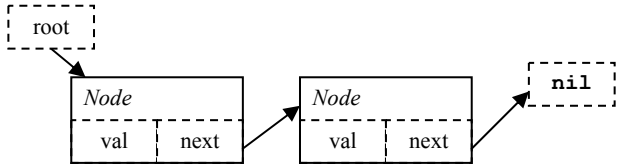


```
   begin i := L; j := R;
      x := a[(L + R) div 2];
      repeat
         while a[i] < x do i := i + 1; end;
         while x < a[j] do j := j - 1; end;
         if i <= j then
             w := a[i];
          a[i] := a[j];
          a[j] := w;
          i := i + 1;
          j := j - 1;
         end;
      until i > j;
      if L < j then QuickSort(a, L, j) end;
      if i < R then QuickSort(a, i, R) end;
   end QuickSort;

(* Заполнение массива числами*)
   procedure FillTheArray;
   var i:integer;
   begin
     for i := 0 to MAX_SIZE - 1 do
       MyArray[i] := abs( 10 - i );
     end;
   end FillTheArray;

(*Печать массива*)
   procedure PrintTheArray;
   var i:integer;
   begin
      writeln("Array:");
      for i := 0 to MAX_SIZE - 1 do
         write( MyArray[i]:2, ', ' );
      end;
      writeln;
   end PrintTheArray;

(*Вызов сортировки*)
   procedure Execute;
   begin
```

```
      QuickSort(MyArray, 0, MAX_SIZE - 1);
    end Execute;
begin
 writeln("Example 2.10 (Quick sort)");
 FillTheArray;
 PrintTheArray;
 Execute;
 PrintTheArray;
 writeln("Type 'q' to exit");
end Example91.
```

11.2. Организация связанного списка

Учитывая особенности языка Zonnon и платформы .NET, рассмотрим реализацию простейшего контейнера, основанного на связанных списках, на языке Zonnon.



Стек – это список, в котором добавление и изъятие элементов осуществляется с одной стороны. Реализовывать его будем при помощи односвязного списка, как показано на рис. выше.

```
module Stack;

record {ref, public} Node;
   val: integer;
   next: Node;
end Node;

var root: Node;

procedure push(n: integer);
var
   newnode: Node;
begin
```



```
   newnode := new Node;
   newnode.val := n;
   newnode.next := root;
   root := newnode;
end push;

procedure pop():integer;
var t: integer;
begin
   if root = nil then return 0; end;
   t := root.val;
   root := root.next;
   return t;
end pop;

begin
   push(3);
   push(7);
   push(10);
   writeln(pop());
   writeln(pop());
   writeln(pop());
end Stack.
```

12. Модульное программирование на Zonnon

Возможность программировать и отлаживать программу по частям в Zonnon поддерживается на уровне модулей. Чтобы использовать в одном модуле типы или процедуры другого модуля, требуется воспользоваться инструкцией **import**.

Формат импорта:

import <что> **as** <синоним>;

Синоним можно не задавать:

import <что>;

Пример:

```
module MyModule1;
procedure {public} PrintHello;
begin
```

```
    writeln("Zonnon Language - Your Language");
end PrintHello;
end MyModule1.

module MyModule;
import MyModule1;

begin
    MyModule1.PrintHello;
end MyModule.
```

Инструкция **import** позволяет импортировать модули, объекты и дефиниции. Для получения доступа к функции требуется импортировать область, в которой она определена.

13. Компилятор ETH Zonnon

Первый компилятор языка Zonnon разработан для платформы Microsoft .NET. Реализация выполнена в Швейцарском федеральном технологическом институте в Цюрихе (ETH Zurich) [5]. В настоящее время доступен только компилятор в версии командной строки; интеграция компилятора в среду разработки Visual Studio .NET ожидается в 2004 году.

В данном разделе приводятся сведения о синтаксисе и параметрах вызова компилятора Zonnon из командной строки.

При установке системы Zonnon компилятор помещается в каталог, заданный пользователем в процессе установки. По умолчанию это c:\Program Files\... Имя выполняемого файла с Zonnon-компилятором – ETH.Zonnon.CommandLineCompiler.exe.

Общий формат вызова компилятора следующий:
ETH.Zonnon.CommandLineCompiler.exe <параметры и исходные файлы>...

Параметры и имена исходных файлов указываются в произвольном порядке. В качестве имен можно указывать как абсолютные, так и относительные имена файлов. Параметры начинаются с символа наклонной черты, непосредственно после которой задается имя параметра. Если параметр подразумевает некоторую дополнительную информацию, то после имени ставится двоеточие, после которого задается эта информация.

13.1. Имена исходных файлов

Имена исходных файлов, в отличие от параметров компиляции, задаются непосредственно, без наклонной черты и специального слова. В качестве имен необходимо указать абсолютное (путь к файлу + имя файла) или относительное имя, например:

```
c:\testsuite\declarations\test5.znn
..\random.znn
```

Если имя исходного файла содержит пробелы, оно должно быть заключено в кавычки, например:

```
"file 20.znn"
“c:\My Programs\My Tests\test05.znn”
```

Можно задавать несколько исходных файлов. Порядок задания несущественен. Компилятор обрабатывает исходные файлы в порядке их задания и всегда генерирует одну результирующую сборку, содержащую объектный код, соответствующий всей совокупности исходных файлов.

13.2. Результат компиляции: параметр /out

Формат параметра:

```
/out:assembly-name-without-extension
```

Параметр используется для задания имени выходного файла. Если этот параметр не указать, то будет создан файл с именем, совпадающим с именем первого по порядку исходного файла. Обратите внимание, что расширение имени файла (.exe или .dll) можно не указывать: компилятор сам добавит необходимое расширение в имя, заданное в параметре, в зависимости от режима компиляции (определяемого посредством параметров /exe и /entry, см. ниже).

13.3. Вид и поведение результирующего кода: параметр /exe

Формат параметра:

/exe

Данный параметр (вместе с параметром /entry, см. ниже) задает вид объектного кода, генерируемого компилятором.

Если параметр /exe задан при запуске компилятора, то он генерирует выполняемую программу (exe-файл). При своем запуске эта программа инициализирует модули (выполняет части begin … end для всех модулей, в которых такие части имеются), после чего инициирует диалог в командном окне. Посредством этого диалога пользователь может вызвать на выполнение любую `public` процедуру без параметров из любого модуля программы.

Диалог поддерживает следующие четыре команды:

<имя модуля>.<имя процедуры>

Команда вызывает на исполнение процедуру без параметров <имя процедуры> из модуля <имя модуля>.

list (или l)

Команда выводит в окно список всех публичных процедур без параметров из всех модулей программы

help (или h или ?)

Команда выводит в окно краткую подсказку.

exit (или quit или q)

Команда завершает диалог и закрывает окно.

Если ни один из параметров – ни /exe, ни /entry – не задан, то компилятор генерирует в качестве результата динамически подключаемую библиотеку Windows (файл с расширением .dll), которую можно использовать из других программ.

Dll-файл, созданный компилятором Zonnon, можно использовать по крайней мере двумя способами:

а) Ссылаться на публичные компоненты файла (модули, объекты и т.д.) из других программ – написанных как на Zonnon, так и на

любых других языках платформы .NET. Для задания ссылок на dll-файлы, ранее созданные компилятором Zonnon, можно использовать параметр /ref, см. ниже.

б) Воспользоваться специальной утилитой dialogue.exe из комплекта поставки Zonnon. Эта утилита открывает командный диалог, посредством которого можно выполнить любую публичную процедуру без параметров из заданного dll-файла. Синтаксис и набор команд утилиты идентичны аналогичной функциональности для случая задания параметра /exe, см. выше.

Замечание 1

Параметры /exe и /entry являются взаимоисключающими; для одного сеанса компиляции может быть задан только один из них.

Замечание 2

Каждый из параметров /exe или /entry может быть задан в командной строке не более одного раза.

13.4. Точка входа в программу: параметр /entry

Параметр имеет следующий формат:

/entry:<имя начального модуля>

Команда задает имя модуля программы, с которой должно начаться ее выполнение. Этот параметр реализует модель выполнения программы, традиционную для языков C/C++/C#/Java. При этом имя начального модуля может быть произвольным, то есть не обязательно main или Main, как в упомянутых языках.

Если задан параметр /entry, то компилятор генерирует выполняемый файл (exe-файл). Параметр /entry несовместим с параметром /exe и может быть задан не более одного раза.

13.5. Режим обработки исключительных ситуаций: параметр /safe

Параметр имеет следующий формат:

/safe

Параметр задает способ обработки исключительных ситуаций в генерируемой программе. Если параметр не задан, то при исполнении

сгенерированной программы исключительная ситуация, для которой не предусмотрен обработчик, приводит к аварийному завершению программы с выдачей стандартного для .NET сообщения, которое включает вывод состояния стека на момент завершения.

Если параметр /safe задан, то при возникновении необработанного исключения выводится лишь стандартное сообщение без распечатки стека.

Замечание: параметр /safe имеет смысл только совместно с параметрами /exe или /entry.

13.6. Ссылки на внешние библиотеки: параметр /ref

Параметр имеет следующий формат:

/ref:<полный путь к библиотечному файлу>

Этот параметр задает имя библиотечного файла (в терминологии .NET – сборки, assembly), ресурсы которого используются в компилируемой программе (посредством объявлений импорта). Синтаксис и семантика параметра идентичны аналогичному параметру компилятора C#.

Обратите внимание, что язык изначального написания библиотеки, на которую ссылается Zonnon-программа (C#, Visual Basic, J# и т.д.), не имеет никакого значения. Все, что необходимо, – чтобы файл, имя которого задано в параметре, содержал корректную .NET-сборку.

Допускается задание нескольких параметров /ref, при этом они не обязательно должны следовать друг за другом.

13.7. Вывод заголовка компилятора: параметр /quiet

Параметр имеет следующий формат:

/quiet

Если параметр задан, то стандартный заголовок компилятора с его названием, номером версии и информацией об авторских правах выводиться не будет.

Литература

1. Gutknecht J., Zueff E. Zonnon Language Report. Zurich: ETH Zentrum, 2003 (www.zonnon.ethz.ch).
2. Microsoft Developer Network Library. Microsoft, 2004 (http://msdn.microsoft.com).
3. Wirth N. Oberon Language Report. Zurich: ETH Zentrum, 1990 (www.oberon.ethz.ch).
4. Zonnon Language – Coding Guidelines. Zurich: ETH Zentrum, 2004.
5. Zueff E. Zonnon Compiler: Architecture, Integration, Technology. Zurich: ETH Zentrum, 2003.
6. Вирт Н. Алгоритмы и структуры данных. СПб.: Невский Диалект, 2001.
7. Сайт проекта Oberon. Zurich: ETH Zentrum (www.oberon.ethz.ch).
8. Сайт проекта Zonnon в ННГУ. Н.Нов.: Лаборатория «Информационные технологии» ННГУ (www.itlab.unn.ru).

Содержание

Для заметок