Е. В. Касьянова

ЯЗЫК ПРОГРАММИРОВАНИЯ ZONNON ДЛЯ ПЛАТФОРМЫ .NET*

ВВЕДЕНИЕ

Данная статья описывает состояние пока еще не завершенной работы над языком, получившим название языка Zonnon [1]. Это новый универсальный язык программирования в семействе языков Паскаль, Модула-2 и Оберон. Он сохраняет стремление к простоте, ясному синтаксису и независимости концепций, а также уделяет внимание параллельности и легкости композиции и выражения. Унификация абстракций является стержнем проектирования языка Zonnon, и она отражается в его концептуальной модели, основанной на модулях, объектах, определениях и реализациях. Язык Zonnon содержит такие новые черты, как активность в объектах, основанный на межобъектном взаимодействии диалог, перегрузка операций и обработка исключительных ситуаций. Язык Zonnon специально разрабатывается как платформно-независимый язык.

Сам проект по разработке Zonnon языка возник как продолжение работы его авторов над языком Оберон (Oberon) в проекте 7, который был начат Microsoft Research в 1999 году с целью реализации значительного числа нестандартных языков программирования для платформы .NET [2]. Язык Оберон [3, 4] является хорошо известным преемником языков Паскаль (Pascal) и Модула-2 (Modula-2). Мотивация авторов на продолжение работы по развитию языка Оберон связана со следующими двумя целями [5]:

a) экспериментально исследовать потенциальные возможности платформы .NET в комбинации с новой технологией CCI интеграции компиляторов для проектирования языков;
b) реализовать для платформы .NET язык Zonnon, являющийся эволюцией языка Oberon для .NET.

Поскольку язык Zonnon задуман как дальнейшая эволюция языка Оберон, авторы стремятся сохранить такие важные черты языка Оберон и его преемников, как компактность языка, ясность, недвусмысленность и ортогональность его основных понятий. Вместе с тем, чтобы создать современную альтернативу языку Оберон, авторы внесли в язык ряд изменений, основными из которых являются:

* Работа выполнена при финансовой поддержке Министерства образования РФ (грант № E02-1.0-42) и компании Майкрософт.

- более развитая модульная структура языка;
- продвинутая и одновременно простая и ясная объектная модель;
- концепция активных объектов.

«Общеалгоритмическая» часть языка Zonnon практически полностью повторяет соответствующие части Оберона, поэтому данный язык можно рассматривать как естественную замену Оберону там, где последний традиционно используется для обучения программированию.

Язык Zonnon возник как естественный результат исследований, проводившихся в течение последних нескольких лет в Федеральном Технологическом Институте (ETH) в Цюрихе. Непосредственными предшественниками языка следует считать Active Oberon [6], реализованный как базовый язык ядра операционной системы BlueBottle, а также реализацию языка Oberon для платформы .NET (Oberon.NET). Язык Zonnon, имея ряд общих черт с указанными языками, вместе с тем, существенно отличается от них по ряду принципиальных моментов. Первая реализация Zonnon выполняется для платформы .NET. Кроме того, предполагается интеграция компилятора в систему программирования Visual Studio .NET (в сотрудничестве с компанией Microsoft).

Статья начинается с изложения основных особенностей языка Zonnon (разд. 1). Разд. 2 посвящен вопросам отображения языка Zonnon на платформу .NET. Разд. 3 содержит описание разрабатываемого компилятора с языка Zonnon для платформы .NET. Полный синтаксис языка Zonnon в его состоянии на август 2003 г. приведен в приложении.

1. ЯЗЫК ПРОГРАММИРОВАНИЯ ZONNON

Язык Zonnon — это универсальный императивный язык программирования. Он характеризуется богатой и мощной, но сильно унифицированной объектной моделью, которая поддерживает разнообразные стили программирования, включая обычный стиль алгоритмов и структур данных, стили модульного и объектно-ориентированного программирования, а также модели вычислений, основанные на агентах (actor-based computing models). Основными чертами объектной модели являются

- унифицированная концепция абстракции, так называемое определение (*definition*),
- соответствующее понятие реализации (*implementation*) по умолчанию,
- комбинация обычных объекта и нити (*thread*), называемая активным объектом (*active object*),

• конструкция модуля (*module*).

Нестрого говоря, определения относят к одной категории и унифицируют общие абстракции суперкласса и интерфейса и в комбинации с механизмом статического агрегирования реализаций заменяют концепции иерархии классов и единственного наследования. Активные объекты появляются вместе с интегрированной нитью управления, которая описывает их поведение во время исполнения. Модули — это объекты с управляемой системой жизненным циклом. Кроме того, к языку добавлен оператор блока (*block statement*) с факультативными модификаторами обработки в фигурных скобках и предложением перехвата исключений. Типичными модификаторами обработки являются ACTIVE (исполнять как отдельный поток), EXCLUSIVE (исполнять при взаимном исключении с соответствующей областью действия объекта) и CONCURRENT (все операторы потенциально могут исполняться параллельно). Ниже данные новые языковые понятия будут проиллюстрированы на небольшом наборе простых, но типичных примеров, приведенных авторами языка [3].

1.1. Определения и реализации

Патефон-автомат имеет две “грани” (“facets”): его можно воспринимать либо как хранилище записей, либо как проигрыватель. Соответственно, можно использовать следующие определения:

```
DEFINITION Store;
PROC Clear;
Add (s: Lib.Song);
END Store.
DEFINITION Player;
VAR cur: Lib.Song;
PROC Play (s: Lib.Song);
PROC Stop;
END Player.
```

Предположим, что, кроме того, имеется следующая реализация `Store` по умолчанию:

```
IMPLEMENTATION Store;
VAR rep: Lib.Song;
PROC Clear;
BEGIN rep := NIL
```

```
END Clear;
PROC Add (s: Lib.Song);
BEGIN s.next := rep; rep := s
END Add;
BEGIN Clear
END Store.
```

Тогда можно использовать следующее определение объекта патефона-автомата:

```
OBJECT JukeBox IMPLEMENTS Player, Store;
IMPORT Store; (* aggregate *)
PROCEDURE Play (s: Lib.Song);
IMPLEMENTS Player.Play;
PROCEDURE Stop IMPLEMENTS Player.Stop;
..
END JukeBox.
```

Заметим, что реализация `Store` по умолчанию неявно агрегируется с пространством объектного состояния.

1.2. Активные объекты

В некотором простом террариуме поддерживается следующий вид поведения содержащихся в нем живых тварей. Если температура опускается ниже некоторого определенного минимума, представители всех видов погружаются в спячку, иначе они либо перемещаются случайным образом, либо, если голодны, требуют помощи.

```
OBJECT Creature;
VAR X, Y, temp, hunger, kill: INTEGER;
PROCEDURE NEW (x, y, t: INTEGER);
BEGIN X := x; Y := y; temp := t;
hunger := 0
END;
PROCEDURE SetTemp (dt: INTEGER);
BEGIN { EXCL } temp := temp + dt
END SetTemp;
BEGIN { ACTIVE }
LOOP
AWAIT temp >= minTemp;
```

```
WHILE hunger > minHunger DO
HuntStep(5, kill);
hunger := hunger – kill;
WHILE (kill > 0) & (hunger > 0) DO
HuntStep(7, kill);
hunger := hunger – kill
END;
RandStep(2)
END;
RandStep(4); hunger := hunger + 1
END
END Creature;
```

Заметим, что тело определения объекта последовательно описывает полную историю жизни таких живых тварей. Также заметим, что их поведение еще зависит существенным образом от среды, вызывающей метод `SetTemp`. В частности, любое существо, находящееся в террариуме, может быть заблокировано оператором WAIT до тех пор, пока температура не поднимется выше минимума.

1.3. Модули

Модули — это объекты системного уровня, чей жизненный цикл управляется системой автоматически. В частности, модуль динамически загружается в момент первого вызова. Модули относятся к “статическим” объектам, которые статически могут импортировать другие модули. Хорошим примером системно-ориентированных модулей является менеджер ресурсов. Следующий набросок показывает менеджера окнами с инкапсулированной структурой данных, которая представляет текущую конфигурацию окна в дисплейном пространстве системы. Заметим, что менеджер окнами содержится в пространстве имен, называемом *System*, и что он базируется на другом модуле, называемом `DisplayManager`.

```
MODULE System.WindowManager;
IMPORT System.DisplayManager;
(* static import *)
OBJECT { VALUE } Pos;
VAR X, Y, W, H: INTEGER
END Pos;
DEFINITION Window;
VAR pos: Pos;
```

```
PROCEDURE Draw ();
END Window;
VAR { PRIVATE } W, H: INTEGER;
bot: OBJ { Window };
PROCEDURE Open(this:OBJECT{Window},p:Pos);
BEGIN ...
END Open;
PROCEDURE Change(this:OBJECT{Window},p:Pos);
BEGIN ...
END Change;
BEGIN (* module initialization *) bot:=NIL;
W := System.DisplayManager.Width();
(* delegation *)
H := System.DisplayManager.Height();
END WindowManager.
```

Структурно унифицировано можно представлять систему периода исполнения как ациклическую иерархию модулей. Обычно внизу и наверху иерархии расположены системные модули и модули приложений соответственно.

2. ОТОБРАЖЕНИЕ ЯЗЫКА ZONNON НА ПЛАТФОРМУ .NET

Для любого языка необходимым условием возможности его реализации для .NET является существование отображения его конструкций на общеязыковую исполняющую среду CLR (Common Language Runtime) [2]. В зависимости от парадигмы и модели, представленных языком, это может создавать серьезную проблему. В данном случае императивного языка отображение исполнимой части на механизм исполнения CLR осуществляется непосредственным образом. Что нужно уточнить по существу — это специфицировать отображение определений, реализаций, активных объектов и модулей.

2.1. Отображение определений и реализаций

Существуют различные опции отображения. Если переменные состояний в определениях отображаются в *properties* или *virtual fields* (пусть они существуют), полное пространство состояний может теоретически синтезироваться в реализующем объекте, но с некоторой потерей эффективности. В отличие от этого решение по отображению на C# (канонический язык для

.NET), приведенное ниже, основывается на внутреннем подручном классе, который обеспечивает пространство агрегированных состояний.

```
DEFINITION D;
TYPE e = (a, b);
VAR x: T;
PROCEDURE f (t: T);
PROCEDURE g (): T;
END D;
IMPLEMENTATION D;
VAR y: T;
PROCEDURE f (t: T);
BEGIN x := t; y := t
END f;
END D;
```

отображается на

```
interface D_i {
T x { get; set; }
void f(T t); T g (); }
internal class D_b {
private T x_b;
public enum e = (a, b);
public T x {
get { return x_b };
set { x_b = ... } } }
public class D_c: D_b {
T y;
void f(T t) {
x_b = t; y = t; } }
```

2.2 Отображение активных объектов

Отображение активных объектов в большей степени техническая, а не концептуальная проблема. Очевидно, что средства многопоточности платформы .NET [6] должны служить в этом случае как образы активных конструкций языка Zonnon. Ниже приводится «лобовое» решение для отображения оператора AWAIT. Оно основывается на массовом упоминании ожидаемых объектов в конце каждой критической секции. Авторы надеются, что в дальнейшем им удастся уточнить это решение.

```
BEGIN { ACTIVE } S END
Method void body() { S };
Field Thread thread;
NEW(x)
x.thread = new Thread(
new ThreadStart(body));
x.thread.Start()
AWAIT c
while !c { Monitor.Wait(this); }
BEGIN { EXCL } S END
Monitor.Enter(this); S;
Monitor.PulseAll(this);
Monitor.Exit(this);
```

2.3. Отображение модулей

По существу модули просто отобразить на “статические” классы, которые представляют собой классы только со статическими членами. Ниже приведен набросок описания образа при отображении Zonnon-программ на платформу .NET для приведенного в разд. 1.3. менеджера окнами:

```
namespace System {
namespace WindowManager {
public struct Pos { ... };
public class Window { public Pos pos;
public virtual Draw ();
}
public sealed class WindowManager {
private static int W, H; Window bot;
public static Open (Window this;Pos p)
{ ... };
public static Change(Window this;Pos p)
{ ... }
public static void WindowManager () {
...
W :=
System.DisplayManager.DisplayManager.Width();
...
}}
}
```

3. КОМПИЛЯТОР С ЯЗЫКА ZONNON ДЛЯ .NET

Компилятор с языка Zonnon разработан для платформы .NET и работает наверху нее. Компилятор воспринимает Zonnon исходники (единицы компиляции) и производит обычный ассемблерный код платформы .NET, содержащий MSIL код и метаданные.

Имеются две версии компилятора: компилятор командной строки и компилятор, интегрированный в среду Visual Studio [7]. Компилятор реализован на языке C# с использованием Common Compiler Infrastructure framework (см. ниже), он спроектирован и разработан в Microsoft Research, г. Редмонде [5].

3.1. Common Compiler Infrastructure

Общекомпиляторная инфраструктура CCI (Common Compiler Infrastructure) — это множество ресурсов программирования (C# классы), обеспечивающих поддержку для реализации компиляторов и других языковых инструментов для платформы .NET. Эта поддержка не является полной: не поддержаны некоторые аспекты функциональности компилятора, например, лексический и синтаксический анализ. Но очень важно то, что CCI поддерживает интеграцию в среду MS Visual Studio. Потенциально, возможно достичь полной интеграции компилятора со всеми компонентами VS, такими как редактор, отладчик, менеджер проектов, система онлайновой помощи и т.д.

Среда CCI могла бы рассматриваться как часть всей платформы .NET; пространство имен `Compiler`, содержащее ресурсы CCI, включено в пространство имен `System`. CCI состоит из трех главных частей: промежуточное представление, множество трансфомеров и интеграционная служба.

Промежуточное представление IR (Intermediate Representation) — это богатая иерархия C# классов, представляющих наиболее общие и типичные понятия современных языков программирования. IR иерархия основывается на архитектуре языка C#: ее классы отражают все C# и CLR понятия, такие как класс, метод, оператор, выражение и т.д. Это позволяет разработчику компилятора представлять напрямую подобные понятия его языка. В случае, когда язык имеет понятия или конструкции, которые не представлены множеством IR классов, есть возможность расширять исходную IR иерархию добавлением новых IR классов. При этом должны быть добавлены также соответствующие трансформации — либо как расширения стандартных “визитёров” (см. ниже), либо в виде полностью нового визитёра.

Трансфомеры (“визитёры”) представляют собой множество основанных классов, выполняющих последовательные преобразования из IR в ассемблерный код платформы .NET. Имеются пять стандартных визитёров, предопределенных в CCI: наблюдатель (Looker), описатель (Declarer), решатель (Resolver), проверяющий (Checker), и нормализатор (Normalizer). Все визитёры обходят IR, выполняя преобразования различных видов. Визитёр наблюдатель (в компании с визитёром-описателем) заменяет вершины Identifier на числа/локалы, к которым они сводятся. Визитёр-решатель разрешает перегрузки операций и выводит типы результатов выражений. Проверяющий визитёр выявляет семантические ошибки и пытается устранить их, а визитёр-нормализатор готовит IR для выстраивания в MSIL и метаданные.

Все визитёры реализованы как классы, наследуемые от класса StandardVisitor из CCI. Допускается как расширение функциональности некоторого визитёра путем добавления методов, обрабатывающих специфические языковые конструкции, так и создание некоторого нового визитёра. CCI требует, чтобы все визитёры, используемые в компиляторе, наследовались (прямо или косвенно) от класса StandardVisitor.

Интеграционная служба представляет собой разнообразные классы и методы, обеспечивающие интеграцию в среду Visual Studio. Указанные классы инкапсулируют специфические структуры данных и функциональность, необходимые для редактирования, отладки, фоновой компиляции и т.д.

3.2. Архитектура компилятора с языка Zonnon

Концептуально организация компилятора вполне традиционна: сканер (Scanner) осуществляет преобразование исходного текста в лексическую свертку, которая воспринимается парсером (Parser). Парсер осуществляет синтаксический анализ и строит абстрактное синтаксическое дерево (AST), используя для входной единицы компиляции классы CCI IR. Каждая вершина AST является экземпляром класса IR. “Семантическая” часть компилятора состоит из серии последовательных преобразований AST, построенного парсером. Результатом таких трансформаций является ассемблерный код платформы .NET, который и является результатом работы компилятора.

Однако с архитектурной точки зрения компилятор с языка Zonnon отличается от большинства “традиционных” компиляторов. Вместо использования технологии “черного ящика”, когда все алгоритмы компилятора и структуры данных инкапсулируются в компиляторе и не видны извне, ком-

пилятор с языка Zonnon, по существу, представляет собой открытый набор ресурсов. В частности, такие структуры данных, как лексическая свертка и дерево AST доступны (через специальный интерфейс) извне компилятора. Аналогичное свойство имеет место для компонентов компилятора: например, возможно вызвать сканер с тем, чтобы извлечь лексемы из конкретной части входа, а затем вызвать парсер, чтобы построить поддерево для этой части входа.

Такая архитектура поддерживается схемой CCI и обеспечивает глубокую и естественную интеграцию компилятора в среду Visual Studio. Для того чтобы поддержать интеграцию, CCI содержит прототипные классы для сканера и парсера. Фактические реализации компонентов сканера и парсера компилятора с языка Zonnon представляют собой классы, наследуемые от этих прототипных классов.

Компилятор расширяет множество вершин IR путем добавления большого числа конкретных типов вершин для понятий Module, Definition, Implementation, Object и для некоторых других конструкций языка Zonnon, которые не имеют прямых прототипов среди CCI-вершин. Дополнительные вершины обрабатываются расширенным визитёром-наблюдателем, который наследует стандарный класс Looker из CCI. Результат работы расширенного наблюдателя семантически эквивалентен дереву AST, содержащему только предопределенные типы CCI. Следовательно, расширенный наблюдатель реализует отображение, описанное в разд. 2.

ЗАКЛЮЧЕНИЕ

В статье представлен язык программирования Zonnon, работа над которым ведется в Цюриховском институте информатики. Разрабатываемый язык задуман как дальнейшая эволюция хорошо известного языка Оберон, являющегося преемником языков Паскаль и Модула-2. Развивая язык Оберон, исходя из современных потребностей в программировании, авторы стремятся сохранить такие важные черты Оберона и его предшественников, как компактность языка, ясность, недвусмысленность и ортогональность его основных понятий. Поэтому можно ожидать, что язык будет востребован теми учебными заведениями, которые в настоящее время используют Паскаль в качестве языка начального обучения программированию, но имеют желание перейти к более современному курсу программирования, охватывающему концепции языков программирования нового поколения,

таких как Java и C#, но осуществить этот переход плавно, без резкого изменения сложившегося стиля преподавания программирования.

СПИСОК ЛИТЕРАТУРЫ

1. Gutknecht J., Zueff E. Zonnon Language Report. — Zurich: Institute of Computer Systems ETH Zentrum, 2003.
2. Уоткинз Д., Хаммонд М., Эйбрамз Б. Программирование на платформе .NET. — М.: Вильямс, 2003.
3. Wirth N. The programming language Oberon // Software — Practice and Experience. — 1988. — Vol. 18, N 6. — P. 671–690.
4. Wirth N. From Modula to Oberon // Software — Practice and Experience. — 1988. — Vol. 18, N 6. — P. 661– 670.
5. Gutknecht J., Zueff E. Zonnon Language Experiment, or How to Implement a Non-Conventional Object Model for .NET // OOPSLA'02. — Seattle, Washington, 2002.
6. Просиз Дж. Программирование для .NET. — М.: Русская Редакция, 2003.
7. Джонсон Б., Скибо К., Янг М. Основы Microsoft Visual Studio .NET 2003. — М.: Русская Редакция, 2003.

ПРИЛОЖЕНИЕ

Ниже для описания синтаксиса языка Zennon используется Расширенный Бекуса—Наура Формализм (Extended Backus—Naur Formalism, EBNF), который характеризуется следующими свойствами:

- альтернативы разделяются символом |;
- скобки [и] обозначают факультативность выражения в скобках;
- скобки { и } обозначают повторение (возможно 0 раз);
- скобки (и) используются для формирования групп элементов;
- нетерминальные символы начинаются с прописной буквы (например, Statement);
- терминальные символы либо начинаются со строчной буквы (например, letter*)*, либо записывается целиком строчными буквами (например, BEGIN), или представляются в виде строк (например, ":=");
- комментарии начинаются с символа // и продолжаются до конца строки.

```
// Синтаксис языка Zonnon в EBNF
// Версия от 8 августа 2003
// 1. Программа и программные единицы
CompilationUnit = { ProgramUnit "." }.
ProgramUnit = ( Module | Definition | Implementation | Object ).
// 2. Модули
Module = MODULE ModuleName [ ImplementationClause ] ";"
[ ImportDeclaration ]
ModuleDeclarations
( BlockStatement | END ) SimpleName.
ModuleDeclarations = { SimpleDeclaration | NestedUnit ";" }
{ ProcedureDeclaration | OperatorDeclaration }
{ ActivityDeclaration }.
NestedUnit = ( Definition | Implementation | Object ).
ImplementationClause = IMPLEMENTS DefinitionName { "," DefinitionName }.
ImportDeclaration = IMPORT Import { "," Import } ";".
Import = ImportedName [ AS ident ].
ImportedName = ( ModuleName | DefinitionName | ImplementationName | Namespace-
    Name ).
// 3. Определения
Definition = DEFINITION DefinitionName [ RefinementClause ] ";"
[ ImportDeclaration ]
DefinitionDeclarations
END SimpleName.
RefinementClause = REFINES DefinitionName.
DefinitionDeclarations = { SimpleDeclaration } { ProcedureHeading | ActivitySignature
    ";" }.
ActivitySignature =
ACTIVITY ActivityName[ TYPE "{" ProtocolEBNF "}" [ EnumType] END Simple-
    Name] ";".
ProtocolEBNF = Спецификация протокола в EBNF на основе алфавита лексем.
// 4. Реализации
Implementation = IMPLEMENTATION ImplementationName ";"
[ ImportDeclaration ]
Declarations
( BlockStatement | END ) SimpleName.
// 5. Объекты
Object = OBJECT [ ObjModifier ] ObjectName [ FormalParameters ] [ Implementation-
    Clause ] ";"
[ ImportDeclaration ]
Declarations
{ ActivityDeclaration }
( BlockStatement | END ) SimpleName.
```

```
ObjModifier = "{" ident "}". // VALUE or REF; VALUE by default
ActivityDeclaration = ACTIVITY ActivityName [ ImplementationClause ] ";"
Declarations ( BlockStatement | END SimpleName ).
// 6. Объявления
Declarations = { SimpleDeclaration } { ProcedureDeclaration }.
SimpleDeclaration = ( CONST [DeclModifier] { ConstantDeclaration ";" }
| TYPE [DeclModifier] { TypeDeclaration ";" }
| VAR [DeclModifier] { VariableDeclaration ";" }
).
DeclModifier = "{" ident "}". // PUBLIC or PRIVATE or IMMUTABLE
ConstantDeclaration = ident "=" ConstExpression.
ConstExpression = Expression.
TypeDeclaration = ident "=" Type.
VariableDeclaration = IdentList ":" Type.
// 7. Типы
Type = ( TypeName [ Width ] | EnumType | ArrayType | ProcedureType | InterfaceType ).
Width = "{" ConstExpression "}".
ArrayType = ARRAY Length { "," Length } OF Type.
Length = ( Expression | "*" ).
EnumType = "(" IdentList ")".
ProcedureType = PROCEDURE [ ProcedureTypeFormals ].
ProcedureTypeFormals = "(" [ PTFSection { ";" PTFSection } ] ")" [ ":" FormalType ].
PTFSection = [ VAR ] FormalType { "," FormalType }.
FormalType = { ARRAY OF } ( TypeName | InterfaceType ).
InterfaceType = OBJECT [ PostulatedInterface ].
PostulatedInterface = "{" DefinitionName { "," DefinitionName } "}".
// 8. Процедуры и знаки операций
ProcedureDeclaration = ProcedureHeading [ ImplementationClause ] ";" [ ProcedureBody
    ";" ].
ProcedureHeading = PROCEDURE [ ProcModifiers ] ProcedureName [ FormalParameters
    ].
ProcModifiers = "{" ident { "," ident } "}". // PRIVATE, PUBLIC, SEALED
ProcedureBody = Declarations BlockStatement SimpleName.
FormalParameters = "(" [ FPSection { ";" FPSection } ] ")" [ ":" FormalType ].
FPSection = [ VAR ] ident { "," ident } ":" FormalType.
OperatorDeclaration = OPERATOR [ OpModifiers ] OpSymbol [ FormalParameters ] ";"
OperatorBody ";".
OperatorBody = Declarations BlockStatement OpSymbol.
OpModifiers = "{" ident { "," ident } [ "," Priority ] "}"
.| "{" Priority "}".
Priority = ConstExpression.
OpSymbol = string. // Строка из 1, 2 или 3 символов из
// ограниченного множества возможных символов
```

```
// 9. Операторы
StatementSequence = Statement { ";" Statement }.
Statement = [ Assignment
| ProcedureCall
| IfStatement
| CaseStatement
| WhileStatement
| RepeatStatement
| LoopStatement
| ForStatement
| AWAIT Expression
| EXIT
| RETURN [ Expression ]
| BlockStatement
| Send
| Receive
].
Assignment = Designator ":=" Expression.
ProcedureCall = Designator.
IfStatement = IF Expression THEN StatementSequence
{ ELSIF Expression THEN StatementSequence }
[ ELSE StatementSequence ]
END.
CaseStatement = CASE Expression OF
Case { "|" Case }
[ ELSE StatementSequence ]
END.
Case = [ CaseLabel { "," CaseLabel } ":" StatementSequence ].
CaseLabel = ConstExpression [ ".." ConstExpression ].
WhileStatement = WHILE Expression DO StatementSequence END.
RepeatStatement = REPEAT StatementSequence UNTIL Expression.
LoopStatement = LOOP StatementSequence END.
ForStatement = FOR ident ":=" Expression TO Expression [ BY ConstExpression ]
DO StatementSequence END.
BlockStatement = BEGIN [ BlockModifiers ]
StatementSequence
{ ExceptionHandler }
[ CommonExceptionHandler ]
END.
BlockModifiers = "{" ident { "," ident } "}". // LOCKED, CONCURRENT, BARRIER
ExceptionHandler = ON ExceptionName { "," ExceptionName } DO StatementSequence.
CommonExceptionHandler = ON EXCEPTION DO StatementSequence.
Send = [Designator] "!" Expression.
```

```
Receive = [Designator] "?" Designator.
// 10. Выражения
Expression = SimpleExpression
[ ( "=" | "#" | "<" | "<=" | ">" | ">=" | IN ) SimpleExpression ]
| Designator IMPLEMENTS DefinitionName
| Designator IS TypeName.
SimpleExpression = [ "+"|"" ] Term { ( "+" | "" | OR ) Term }.
Term = Factor { ( "*" | "/" | DIV | MOD | "&" ) Factor }.
Factor = number
| CharConstant
| string
| NIL
| Set
| Designator
| NEW TypeName [ "(" ActualParameters ")" ]
| NEW ActivityInstanceName
| "(" Expression ")"
| "~" Factor.
Set = "{" [ SetElement { "," SetElement } ] "}".
SetElement = Expression [ ".." Expression ].
Designator = Instance
| Designator "^" // Dereference
| Designator "[" Expression { "," Expression } "]"// Array element
| Designator "(" [ ActualParameters ] ")" // Function call
| Designator "." MemberName // Member selector
Instance = ( SELF | InstanceName | DefinitionName "(" InstanceName ")" ).
ActualParameters = Actual { "," Actual }.
Actual = Expression [ "{" [ VAR ] FormalType "}" ]. // Аргумент с сигнатурой типа
// 11. Константы
number = (integer | real) [ "{" Width "}" ].
integer = digit { digit } | digit { HexDigit } "H".
real = digit { digit } "." { digit } [ ScaleFactor ].
ScaleFactor = "E" ["+" | "" ] digit { digit }.
HexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
CharConstant = "" character "" | "" character "" | digit { HexDigit } "X".
string = "" { character } "" | "" { character } "".
character = letter | digit | Other.
Other = // Любой символ алфавита, кроме тех, которые используются …
// 12. Идентификаторы и имена
ident = ( letter | "_" ) { letter | digit | "_" }.
letter = "A" | ... | "Z" | "a" | ... | "z" | любая другая "культурно-определенная" буква
IdentList = ident { "," ident }.
```

```
QualIdent = { ident "." } ident.
DefinitionName = QualIdent.
ModuleName = QualIdent.
NamespaceName = QualIdent.
ImplementationName = QualIdent.
ObjectName = QualIdent.
TypeName = QualIdent.
ExceptionName = QualIdent.
InstanceName = QualIdent.
ActivityInstanceName = QualIdent.
ProcedureName = ident.
ActivityName = ident.
MemberName = ( ident | OpSymbol ).
SimpleName = ident.
```