# Визиторы в PascalABC.NET и Roslyn

Захаренко А.С. (4 курс, 9 группа)

# Содержание

- Визиторы в PascalABC.NET
- Визиторы в Roslyn (компилятор C#)

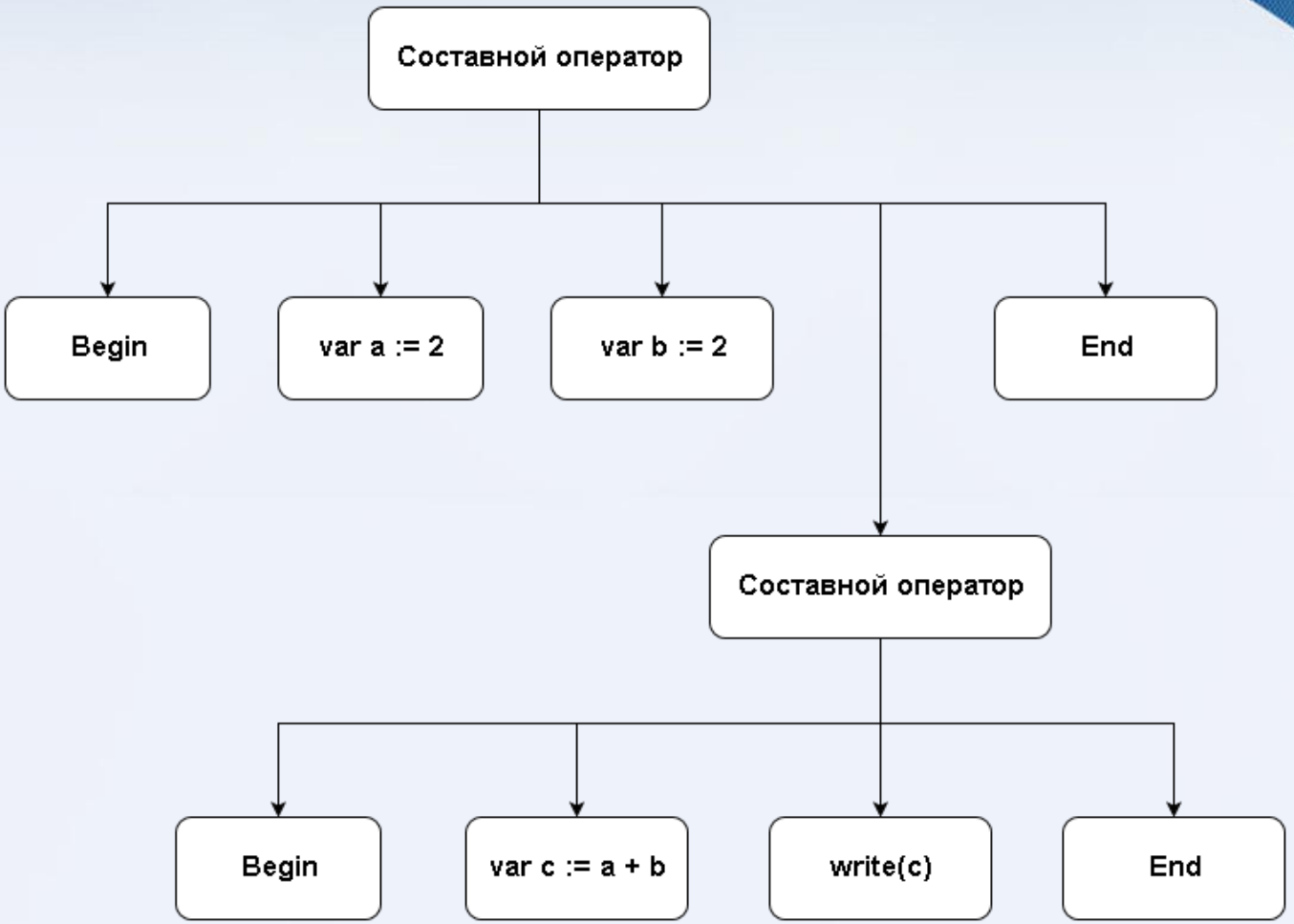
# Содержание

- Визиторы в PascalABC.NET
- Визиторы в Roslyn (компилятор C#)

# Удаление лишних begin-end

Пример программы и соответствующего синтаксического дерева:

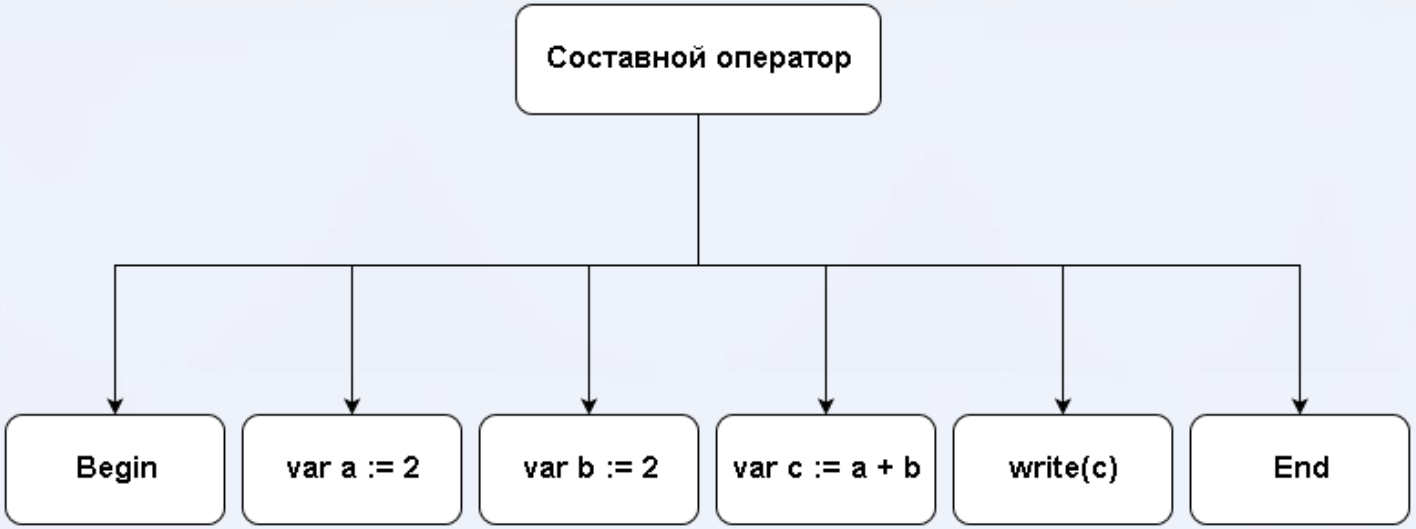
```
begin
  var a := 2;
  var b := 2;
  begin
    var c := a + b;
    write(c);
  end;
end.
```



# Удаление лишних begin-end

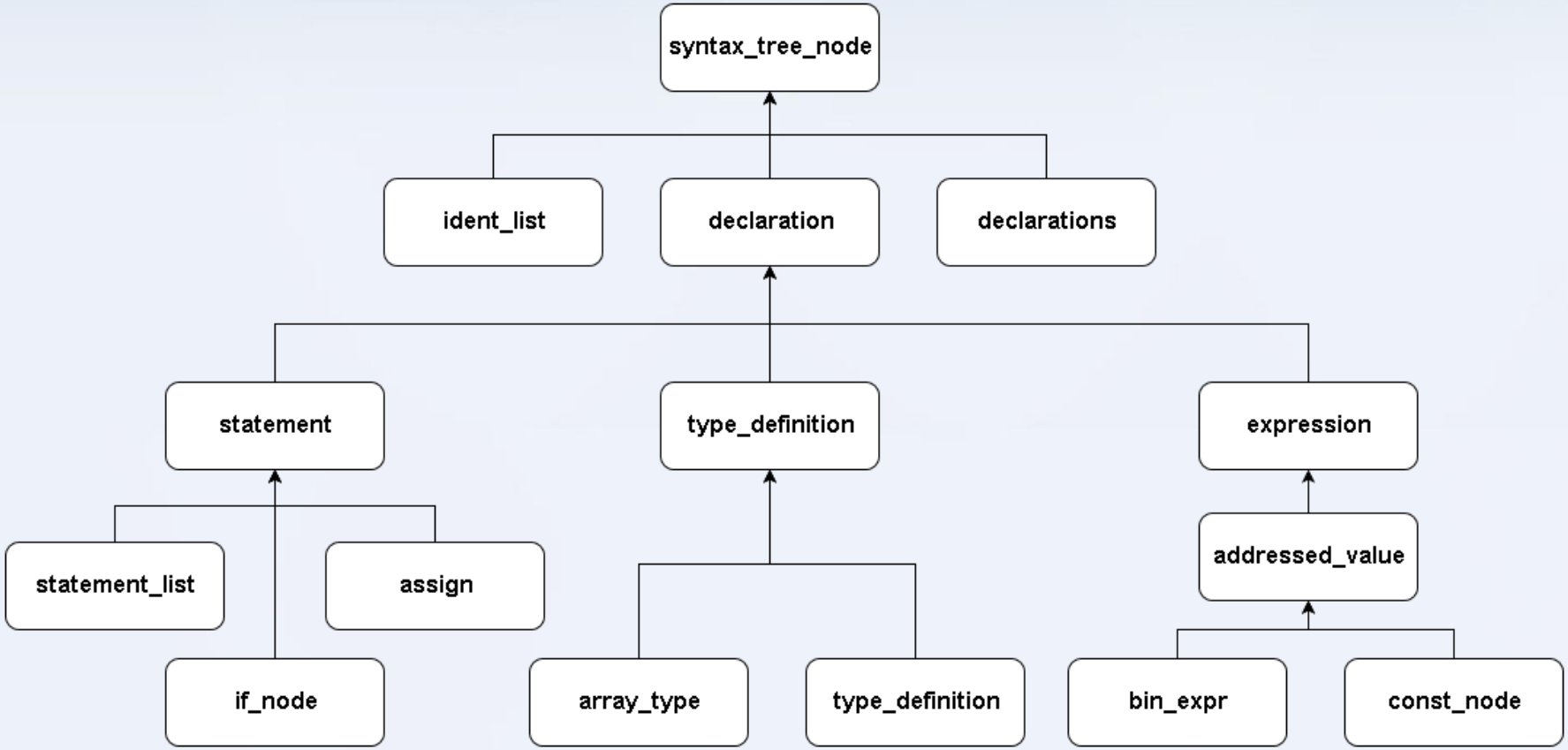
После прохода по дереву визитора на изменение:

```
begin
  var a := 2;
  var b := 2;
  var c := a + b;
  write(c);
end.
```



# Иерархия наследования синтаксических узлов

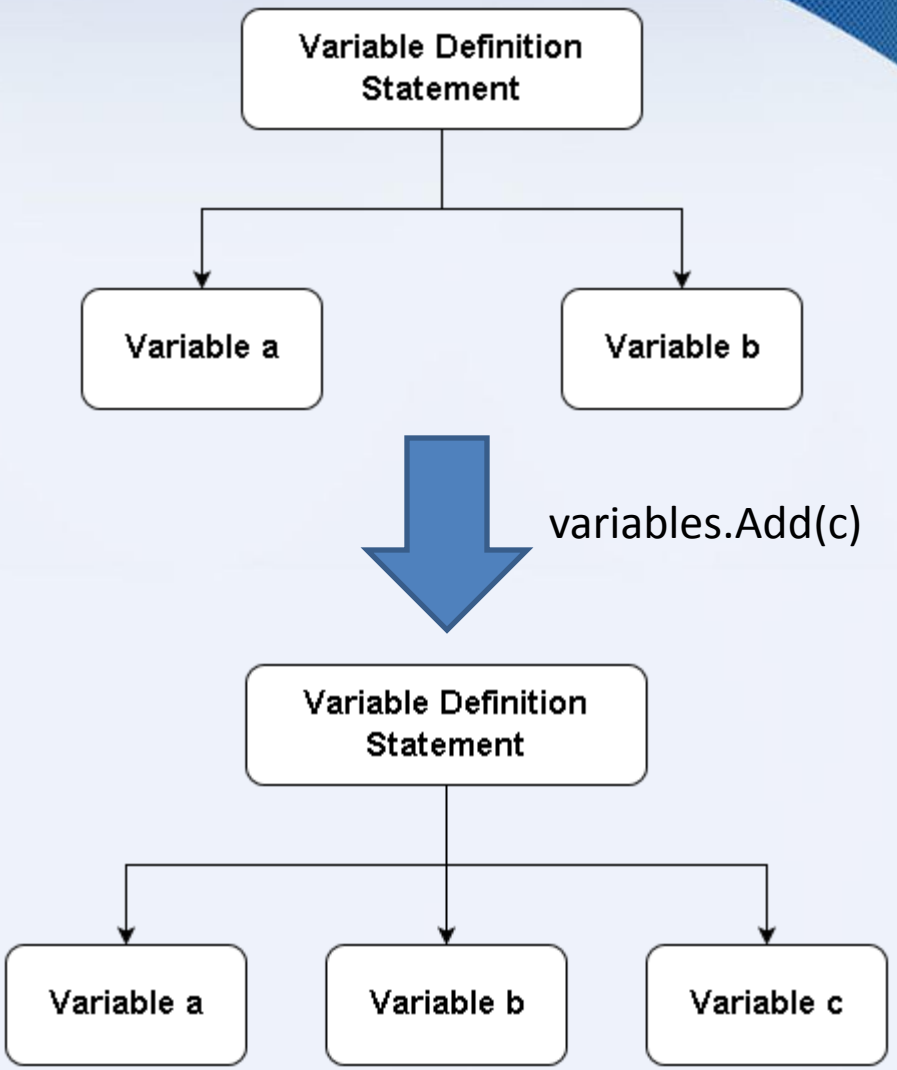
Важнейшие синтаксические узлы:



# Дополнительные методы синтаксических узлов

Функции, сгенерированные для узлов, содержащих списки подузлов :

1. Добавление
2. Удаление
3. Поиск
4. Замена



# Генерация по шаблону

«list_name» и «list_element_type» – переменные шаблона

Пример генерации для класса «список идентификаторов»

```
public void AddFirst(list_element_type el)
{
     list_name.Insert(0, el);
}

private int FindIndex(list_element_type el)
{
     var ind = list_name.FindIndex(x => x == el);
     if (ind == -1)
         throw new Exception(""Элемент не найден"");
     return ind;
}

public bool Remove(list_element_type el)
{
     return list_name.Remove(el);
}
```

```
public void AddFirst(ident el)
{
    idents.Insert(0, el);
}

private int FindIndex(ident el)
{
    var ind = idents.FindIndex(x => x == el);
    if (ind == -1)
        throw new Exception("Элемент не найден");
    return ind;
}

public bool Remove(ident el)
{
    return idents.Remove(el);
}
```

# Иерархия базовых визиторов

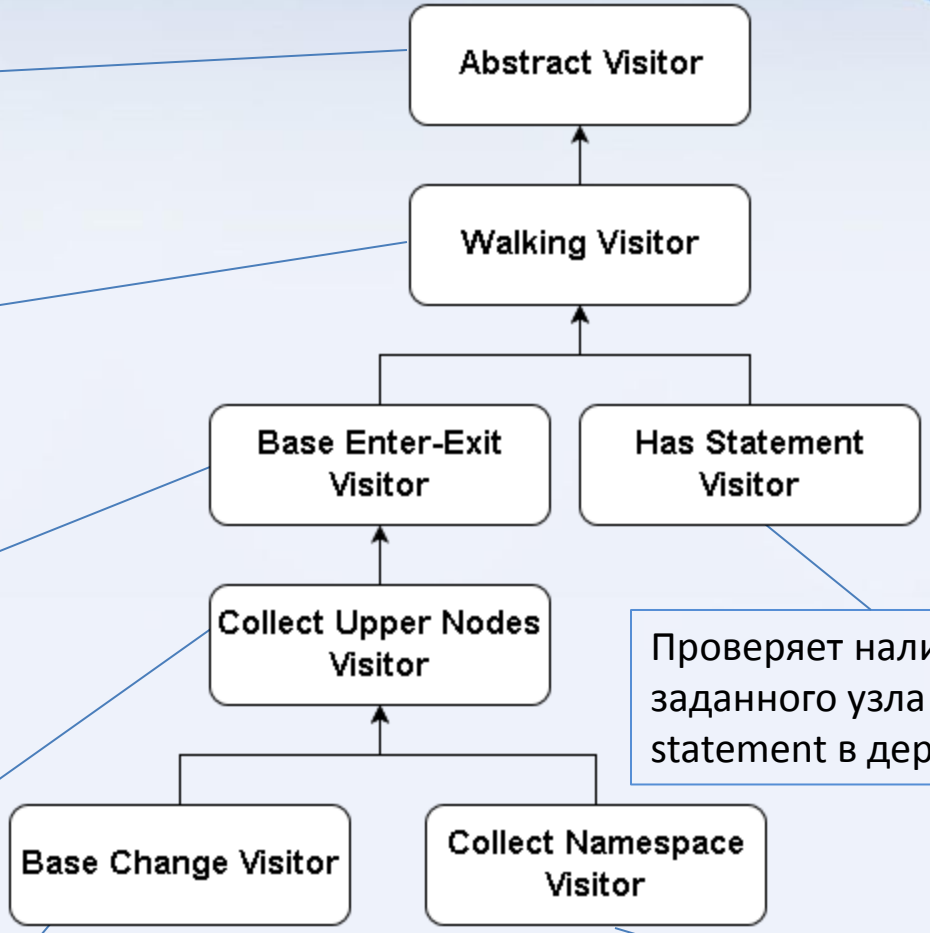
Для каждого узла описаны свои действия, выполняемые при посещении

Содержит делегаты для выполнения действий над узлом при входе и при выходе

Объявляет делегаты переопределяемыми в наследниках методами

Накапливает информацию о предках при обходе узлов

Предоставляет базовые возможности редактирования узлов




Проверяет наличие заданного узла типа statement в дереве

Предоставляет информацию о текущем пространстве имен

# Прикладные визиторы

| Название визитора | Назначение |
|---|---|
| Pretty Printer Visitor | Восстанавливает исходный текст программы с учетом правил форматирования |
| Count Nodes Visitor | Подсчитывает количество узлов каждого типа в дереве |
| Delete Local Defs | Удаляет объявления локальных переменных |
| Delete Redundant Begin-Ends | Удаляет излишние объявления составных операторов |
| Lowering Visitor | Заменяет циклы на условные операторы и безусловные переходы |
| Delete Unused Variables | Удаляет неиспользованные переменные |

# Abstract Visitor

Описание и возможности:

- Обходит все узлы дерева
- Для каждого узла описаны свои действия, выполняемые при посещении
- Позволяет определить действие по умолчанию при обходе узла

```
public virtual void DefaultVisit(syntax_tree_node n)
{
}

public virtual void visit(statement _statement)
{
        DefaultVisit(_statement);
}
```

# Walking Visitor

Описание и возможности:

- Содержит переопределяемые действия, совершаемые перед и после посещения узла
- Позволяет посещать не все узлы (флаг visitNode контроллирует обход узла)

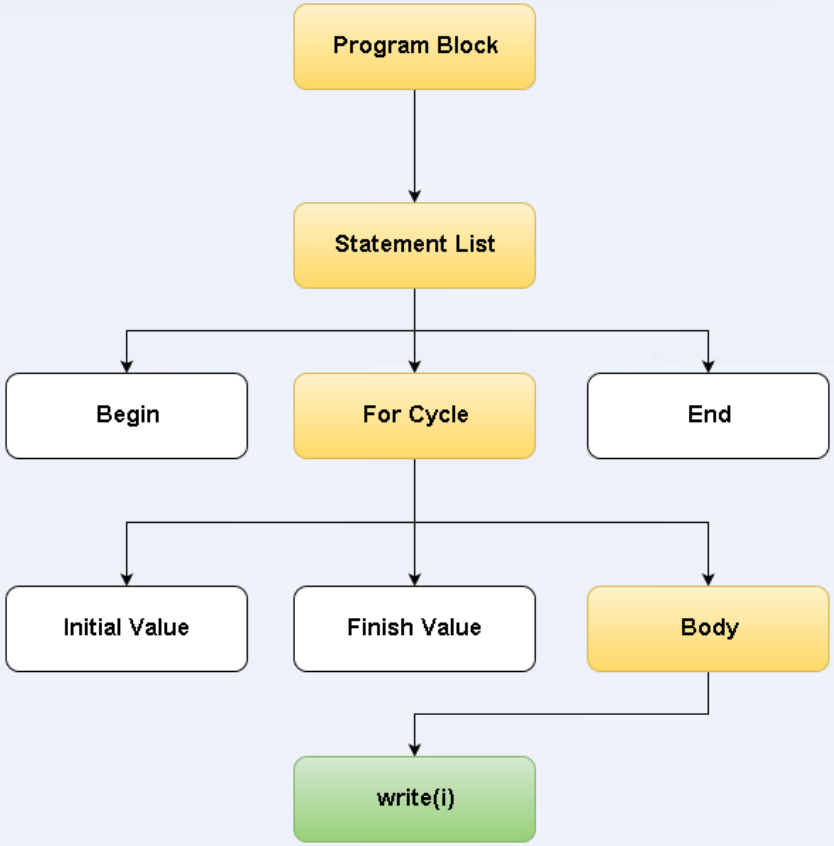
```
public override void DefaultVisit(syntax_tree_node n)
{
        var count = n.subnodes_count;
        for (var i = 0; i < count; i++)
                ProcessNode(n[i]);
}

public virtual void ProcessNode(syntax_tree_node Node)
{
        if (Node != null)
        {
                if (OnEnter != null)
                      OnEnter(Node);

                if (visitNode)
                        Node.visit(this);
                else visitNode = true;

                if (OnLeave != null)
                        OnLeave(Node);
        }
}
```

# Collect Upper Nodes Visitor

```
begin
  for var i := 0 to 5 do
    write(i); <- текущая позиция визитора
end.
```

Визитор накапливает все узлы по пути от текущей позиции до корня.

В данном случае накоплены следующие узлы:




# Base Change Visitor

Часть методов базового визитора на изменение:

```
public void Replace(syntax_tree_node from, syntax_tree_node to)
{
        var upper = UpperNode(); // обращение к предку
        if (upper == null)
                throw new Exception("У корневого элемента нельзя получить UpperNode");
        upper.Replace(from, to); // использование методов, встроенных в узел
}

public T UpperNodeAs<T>(int up = 1) where T : syntax_tree_node
{
        var stl = UpperNode(up) as T;
        if (stl == null)
                throw new Exception("Элемент вложен не в " + typeof(T));
        return stl;
}

public void ReplaceStatement(statement from, IEnumerable<statement> to)
{
        var stl = UpperNodeAs<statement_list>();
        stl.Replace(from, to);
}
```

# Удаление лишних begin-end блоков

Реализация метода Exit

```
public class DeleteRedundantBeginEnds : BaseChangeVisitor
{
    public override void Exit(syntax_tree_node st)
    {
        var stl = st as statement_list;
        if (stl != null && UpperNode() is statement_list)
            ReplaceStatement(stl, stl.subnodes);

        var lst = st as labeled_statement;
        if (lst != null)
        {
            var sttl = lst.to_statement as statement_list;
            if (sttl != null)
            {
                sttl.subnodes[0] = new labeled_statement(lst.label_name, sttl.subnodes[0]);
                ReplaceStatement(lst,sttl.subnodes);
            }
        }

        base.Exit(st);
    }
}
```

# Lowering-визитор

- Замена циклов на условные операторы и метки «goto»
- Переопределение поведения для необходимых узлов
- Наличие фильтра узлов

Переопределение действия, совершаемого перед началом обхода узла. В условном операторе указываются узлы, которые нужно обойти. Для остальных флаг visitNode принимает значение false и они не обходятся визитором:

```
public override void Enter(syntax_tree_node st)
{
        base.Enter(st);
        if (!(st is procedure_definition || st is block || st is statement_list || st is case_node ||
              st is for_node || st is foreach_stmt || st is if_node || st is repeat_node ||
              st is while_node || st is with_statement || st is try_stmt || st is lock_stmt))
                {
                        visitNode = false;
                }
}
```

# Lowering-визитор

Замена проверок ‘is’ на проверку принадлежности множеству

```
private static HashSet<Type> VisitNodes = new HashSet<Type>(
        new[] {
                typeof(procedure_definition),
                typeof(block),
                typeof(statement_list),
                typeof(case_node),
                typeof(for_node),
                typeof(foreach_stmt),
                typeof(if_node),
                typeof(repeat_node),
                typeof(while_node),
                typeof(with_statement),
                typeof(try_stmt),
                typeof(lock_stmt)
        });

public override void Enter(syntax_tree_node st)
{
        base.Enter(st);
        if (!VisitNodes.Contains(st.GetType()))
                visitNode = false;
}
```

# Lowering-визитор

Пример реализации обхода узла while_node с заменой цикла на if и безусловные переходы:

```
public override void visit(while_node wn)
{
        ProcessNode(wn.statements);
        var b = HasStatementVisitor<yield_node>.Has(wn);
        if (!b)
                return;

        var gt1 = goto_statement.New;
        var gt2 = goto_statement.New;

        var if0 = new if_node(un_expr.Not(wn.expr), gt1);
        var lb2 = new labeled_statement(gt2.label, if0);
        var lb1 = new labeled_statement(gt1.label);

        ReplaceStatement(wn, SeqStatements(lb2, wn.statements, gt2, lb1));

        // в declarations ближайшего блока добавить описание labels
        block bl = listNodes.FindLast(x => x is block) as block;

        bl.defs.Add(new label_definitions(gt1.label, gt2.label));
}
```

# Результат работы

Слева – исходный код, справа – результат работы визитора

```
var  i: integer;


begin
  while i < 3 do
  begin
    writeln(i);
    i += 1;
  end;
  writeln('done');
end.
```

```
var  i: integer;
label 1, 2;

begin
  2: if not (i < 3) then
    goto 1;
  writeln(i);
  i += 1;
  goto 2;
  1: writeln('done');
end.
```

# Задача: удалить неиспользуемые переменные

```
var  a: integer;

begin
    var b,c: integer;
    a := 5;
end.
```

# Задача: удалить неиспользуемые переменные

```
var  a: integer;

procedure f;
    var  a: integer;
begin
    var b: integer;
    a := 5
end;

begin
    var b,c: integer;
    a := 5;
end.
```

# Задача: удалить неиспользуемые переменные

```
var  a: integer;
procedure f;
    var  a: integer;
begin
    var b: integer;
    a := 5
end;

type  Person = class
Public
    procedure f;
        var  a: integer;
    begin
    end;
end;

begin
    var b,c: integer;
    a := 5;
end.
```

# Namespace Collector

## Реализация CollectNamespaceVisitor’a

```
public delegate void OnNamespaceChangedDelegate(Namespace newNamespace);

protected OnNamespaceChangedDelegate OnNamespaceChanged;

private Namespace currentNamespace;
protected Namespace CurrentNamespace
{
        get { return currentNamespace; }
        private set
        {
                currentNamespace = value;
                OnNamespaceChanged?.Invoke(currentNamespace);
        }
}

public CollectNamespaceVisitor()
{
        CurrentNamespace = new Namespace("Root");
        OnNamespaceChanged = NamespaceChanged;
}

public virtual void NamespaceChanged(Namespace newNamespace) { }
```

# Namespace Collector

## Переопределение метода Enter

```
public override void Enter(syntax_tree_node st)
{
        base.Enter(st);

        if (st is procedure_definition)
        {
                var procedureDefinition = st as procedure_definition;

                string procedureName =
                        (procedureDefinition.proc_header as function_header)?.name.meth_name.name ??
                        (procedureDefinition.proc_header as procedure_header)?.name.meth_name.name;

                Debug.Assert(procedureName != null);

                CurrentNamespace = new Namespace(CurrentNamespace, procedureName);
        }

        if (st is class_definition)
        {
                string className = (UpperNode() as type_declaration)?.type_name.name;

                Debug.Assert(className != null);

                CurrentNamespace = new Namespace(CurrentNamespace, className);
        }
}
```

# Namespace Collector

Переопределение метода Exit

```
public override void Exit(syntax_tree_node st)
{
        if (st is procedure_definition || st is class_definition)
                CurrentNamespace = CurrentNamespace.ParentNamespace;

        base.Exit(st);
}
```

# Удаление неиспользованных переменных

Реализация класса DeleteUnusedVariables

```
public class DeleteUnusedVariables : CollectNamespaceVisitor
{
        Dictionary<Namespace, HashSet<string>> variablesForRemoving =
                new Dictionary<Namespace, HashSet<string>>();

        public DeleteUnusedVariables()
        {
                variablesForRemoving[CurrentNamespace] = new HashSet<string>();
        }

        public override void NamespaceChanged(Namespace newNamespace)
        {
                if (!variablesForRemoving.ContainsKey(newNamespace))
                        variablesForRemoving[newNamespace] = new HashSet<string>();
        }

        ...
}
```

# Удаление неиспользованных переменных

## Переопределение метода Enter

```
public override void Enter(syntax_tree_node st)
{
        base.Enter(st);

        if (st is dot_node)
        {
                string variableName = ((st as dot_node).left as ident)?.name;
                if (variableName != null)
                        variablesForRemoving[CurrentNamespace].Remove(variableName);

                visitNode = false;
        }

        if (st is var_def_statement)
        {
                HashSet<string> currentVariablesForRemoving = variablesForRemoving[CurrentNamespace];

                foreach (ident ident in (st as var_def_statement).vars.idents)
                        currentVariablesForRemoving.Add(ident.name);

                visitNode = false;
        }
}
```

# Удаление неиспользованных переменных

Переопределение метода visit для идентификатора

```
public override void visit(ident _ident)
{
        Namespace currentNamespace = CurrentNamespace;

        while (currentNamespace != null)
        {
                HashSet<string> currentVariablesForRemoving = variablesForRemoving[currentNamespace];
                if (currentVariablesForRemoving.Contains(_ident.name))
                {
                        currentVariablesForRemoving.Remove(_ident.name);
                        break;
                }

                currentNamespace = currentNamespace.ParentNamespace;
        }
}
```

# Удаление неиспользованных переменных

Переопределение метода Exit

```
public override void Exit(syntax_tree_node st)
{
        if (st is block)
        {
                var unused = variablesForRemoving[CurrentNamespace];
                if (unused.Count > 0)
                {
                        LimitedLocalDefDeleter deleteVisitor = new LimitedLocalDefDeleter(unused, 0);
                        st.visit(deleteVisitor);
                }
        }

        base.Exit(st);
}
```

# Пример работы

Слева – исходный код. Справа – результат работы DeleteUnusedVariables.

```
var  a: integer;
procedure f;
    var  a: integer;
begin
    var b: integer;
    a := 5
end;

type  Person = class
Public
    procedure f;
    var  a: integer;
    begin
    end;
end;

begin
    var b,c: integer;
end.
```

```
procedure f;
    var  a: integer;
begin

    a := 5
end;

type  Person = class
Public
    procedure f;

    begin
    end;
end;

begin

end.
```

# Содержание

- Визиторы в PascalABC.NET
- Визиторы в Roslyn (компилятор C#)

# Иерархия базовых визиторов

Параметризован шаблоном типа выходного значения

Посещает только первый узел, поданый на вход методу Visit.

Визитор для «изменения» синтаксического дерева

Базовый визитор для обхода синтаксического дерева

# C# Syntax Visitor

Базовый визитор узлов

```
public abstract class CSharpSyntaxVisitor
{
        public virtual void Visit(SyntaxNode node)
        {
                if (node != null)
                {
                        ((CSharpSyntaxNode)node).Accept(this);
                }
        }

        public virtual void DefaultVisit(SyntaxNode node)
        {
        }

        public virtual void VisitIdentifierName(IdentifierNameSyntax node)
        {
                this.DefaultVisit(node);
        }

        public virtual void VisitQualifiedName(QualifiedNameSyntax node)
        {
                this.DefaultVisit(node);
        }
}
```

# C# Syntax Visitor

Базовый визитор узлов

```
public abstract class CSharpSyntaxVisitor<TResult>
{
        public virtual TResult Visit(SyntaxNode node)
        {
                if (node != null)
                {
                        return ((CSharpSyntaxNode)node).Accept(this);
                }
                return default(TResult);
         }

         public virtual TResult DefaultVisit(SyntaxNode node)
         {
                return default(TResult);
         }

        public virtual TResult VisitIdentifierName(IdentifierNameSyntax node)
        {
                return this.DefaultVisit(node);
        }

        public virtual TResult VisitQualifiedName(QualifiedNameSyntax node)
        {
                return this.DefaultVisit(node);
        }
}
```

# C# Syntax Walker

Методы класса CSharpSyntaxWalker

```
public abstract class CSharpSyntaxWalker : CSharpSyntaxVisitor
{
        protected SyntaxWalkerDepth Depth { get; }

        protected CSharpSyntaxWalker(SyntaxWalkerDepth depth = SyntaxWalkerDepth.Node)
        {
                this.Depth = depth;
        }

        public override void DefaultVisit(SyntaxNode node)
        {
                ...
        }
}
```

# C# Syntax Walker Depth

Описание типа глубины обхода дерева

```
public enum SyntaxWalkerDepth : int
{
            // Узлы, имеющие потомков
            Node = 0,

            // Узлы, без потомков
            Token = 1,

            // Пробелы, комментарии, директивы
            Trivia = 2,

            // Представляет структуру внутренних частей (например директивы)
            StructuredTrivia = 3,
}
```

# C# Syntax Walker

## Переопределение обхода по умолчанию

```
public override void DefaultVisit(SyntaxNode node)
{
        var childCnt = node.ChildNodesAndTokens().Count;
        int i = 0;

        do
        {
                var child = ChildSyntaxList.ItemInternal((CSharpSyntaxNode)node, i);
                i++;

                var asNode = child.AsNode();
                if (asNode != null)
                {
                        if (this.Depth >= SyntaxWalkerDepth.Node)
                        {
                                this.Visit(asNode);
                        }
                }
                else
                {
                        if (this.Depth >= SyntaxWalkerDepth.Token)
                        {
                                this.VisitToken(child.AsToken());
                        }
                }
        } while (i < childCnt);
}
```

# Node Printer: пример использования Syntax Walker

Перегрузка метода Visit

```
public class NodePrinter : CSharpSyntaxWalker
{
        static int Tabs = 0;
        public override void Visit(SyntaxNode node)
        {
                Tabs++;
                var indents = new String(' ', Tabs * 2);
                Console.WriteLine(indents + node.Kind());
                base.Visit(node);
                Tabs--;
         }
    }
```

# Node Printer: пример использования Syntax Walker

Вызов визитора для конкретного дерева

```
static void Main(string[] args)
{
        var tree = CSharpSyntaxTree.ParseText(@"
        public class MyClass
        {
                public void MyMethod()
                {
                }
                public void MyMethod(int n)
                {
                }
        ");

        var walker = new NodePrinter();
        walker.Visit(tree.GetRoot());
        Console.ReadKey();
}
```

```
CompilationUnit
    ClassDeclaration
      MethodDeclaration
        PredefinedType
        ParameterList
        Block
      MethodDeclaration
        PredefinedType
        ParameterList
          Parameter
            PredefinedType
        Block
```

# C# Syntax Rewriter

Переопределение методов посещения. Метод обновления узла.

```
public override CSharpSyntaxNode VisitExpressionStatement(ExpressionStatementSyntax node)
{
        var expression = (ExpressionSyntax)this.Visit(node.Expression);
        var semicolonToken = (SyntaxToken)this.Visit(node.SemicolonToken);
        return node.Update(expression, semicolonToken);
}

public ExpressionStatementSyntax Update(ExpressionSyntax expression, SyntaxToken semicolonToken)
{
        if (expression != this.Expression || semicolonToken != this.SemicolonToken)
        {
                var newNode = SyntaxFactory.ExpressionStatement(expression, semicolonToken);
                var diags = this.GetDiagnostics();
                if (diags != null && diags.Length > 0)
                        newNode = newNode.WithDiagnosticsGreen(diags);
                var annotations = this.GetAnnotations();
                if (annotations != null && annotations.Length > 0)
                        newNode = newNode.WithAnnotationsGreen(annotations);
                return newNode;
        }

        return this;
}
```

# C# Syntax Rewriter: пример использования

Удаление пустых операторов. Переопределение метода посещения пустого оператора

```
public class EmtpyStatementRemoval : CSharpSyntaxRewriter
{
        public override SyntaxNode VisitEmptyStatement(EmptyStatementSyntax node)
        {
                return null;
        }
}
```

Результат работы визитора:

```
public class Sample
{
        public void Foo()
        {
                Console.WriteLine();
                ;
        }
}
```

```
public class Sample
{
        public void Foo()
        {
                Console.WriteLine();
        }
}
```

# C# Syntax Rewriter: пример использования

Учитываем наличие Trivia

```
public class EmtpyStatementRemoval : CSharpSyntaxRewriter
{
    public override SyntaxNode VisitEmptyStatement(EmptyStatementSyntax node)
    {
        return node.WithSemicolonToken(
            SyntaxFactory.MissingToken(SyntaxKind.SemicolonToken)
                .WithLeadingTrivia(node.SemicolonToken.LeadingTrivia)
                .WithTrailingTrivia(node.SemicolonToken.TrailingTrivia));
    }
}
```

# C# Syntax Rewriter: пример использования

Результат работы улучшенного визитора:

```
public class Sample
{
        public void Foo()
        {
                Console.WriteLine();
                #region SomeRegion
                // Код
                #endregion
                ;
         }
}
```

```
public class Sample
{
        public void Foo()
        {
                Console.WriteLine();
                #region SomeRegion
                // Код
                #endregion

         }
}
```

# Operation Visitor: семантический уровень

Реализация визитора

```
public abstract class OperationVisitor
{
        public virtual void Visit(IOperation operation)
        {
                operation?.Accept(this);
        }

        public virtual void DefaultVisit(IOperation operation)
        {

        }

        public virtual void VisitBlockStatement(IBlockStatement operation)
        {
                DefaultVisit(operation);
        }

        ...
}
```

# Operation Walker: семантический уровень

Реализация визитора

```
public abstract class OperationWalker : OperationVisitor
{
        public override void Visit(IOperation operation)
        {
                operation?.Accept(this);
        }

        public override void VisitVariableDeclaration(IVariableDeclaration operation)
        {
                Visit(operation.InitialValue);
        }

        ...
}
```

# LINQ запросы к дереву

Исходный код

```
public static TSource Do<TSource>(this TSource source, Action<TSource> action)
        where TSource : class
{
        if (source != default(TSource))
        {
                action(source);
        }
        return source;
}
```

# LINQ запросы к дереву

Изменение дерева с использованием запросов

```
SyntaxTree tree = SyntaxTree.ParseText(source);
MethodDeclarationSyntax method = tree.GetRoot()
        .DescendantNodes()
        .OfType<MethodDeclarationSyntax>()
        .First();

IfStatementSyntax ifStatement = method
        .DescendantNodes()
        .OfType<IfStatementSyntax>()
        .First();

method = method.ReplaceNode(ifStatement,
                            ifStatement
        .WithCondition((Syntax.LiteralExpression(SyntaxKind.TrueLiteralExpression))));
```

# LINQ запросы к дереву

## Исходный код

```
public static TSource Do<TSource>(this TSource source, Action<TSource> action)
        where TSource : class
{
        if (source != default(TSource))
        {
                action(source);
        }
        return source;
}
```

## Измененный код

```
public static TSource Do<TSource>(this TSource source, Action<TSource> action)
        where TSource : class
{
        if (true)
        {
                action(source);
        }
        return source;
}
```

# Ссылка на проект в системе github

PascalABC.NET Visitors