

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИИ
**Федеральное государственное автономное образовательное
учреждение высшего образования
ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ**

Институт математики, механики и компьютерных наук
имени И. И. Воровича

Направление подготовки 02.03.02 - Фундаментальная информатика и информа-
ционные технологии

Реализация оператора `yield` по синтаксическому дереву

Выпускная квалификационная работа на степень бакалавра

Студента 4 курса 9 группы

О.А. Баташова

Научный руководитель:

доц. каф. АДМ, к.ф.-м.н. С.С. Михалкович

Допущено к защите:

руководитель направления ФИИТ _____ В. С. Пилиди

Ростов-на-Дону

2016 г.

Оглавление

Введение	4
Постановка задачи.....	5
Исследование предметной области	6
Алгоритмы Lowering.....	8
Развертка конструкции If.....	8
Развертка конструкции while	9
Развертка конструкции Repeat	9
Развертка конструкции For.....	10
Развертка конструкции Foreach	10
Развертка конструкции Case	11
Основные концепции реализации метода-итератора как синтаксического сахара	12
Проблема сохранения состояния метода-итератора.....	12
Архитектура метода-итератора как синтаксического сахара	12
Захват имён	14
Классификация имён.....	14
Алгоритмы захвата имён	15
Захват локальных переменных метода-итератора	15
Захват формальных параметров метода-итератора	17
Захват имен класса, содержащего метод-итератор.....	17
Захват глобальных имен	18
Особенности реализации метода-итератора как синтаксического сахара	19
Особенности Lowering в PascalABC.NET.....	19
Особенности реализации оператора yield в PascalABC.NET	22

Алгоритм преобразования тела метода-итератора в детерминированный конечный автомат.....	23
Тестирование.....	25
Тестирование глобального метода-итератора без захвата переменных	25
Тестирование глобального метода-итератора с захватом локальной переменной.....	26
Тестирование метода-итератора в классе с захватом имени из базового класса	28
Тестирование компиляции метода-итератора со статическим именем в теле	29
Тестирование компиляции метода-итератора с использованием переменной result в теле (негативный тест).....	29
Тестирование компиляции метода-итератора с вложенными подпрограммами (негативный тест)	30
Тестирование компиляции метода-итератора с блоком try..except (негативный тест).....	30
Приложения	31
Код развертки глобального метода-итератора с захватом локальной переменной.....	31
Заключение.....	33
Список использованной литературы.....	34

Введение

В современном программировании часто встречается задача реализации пользовательского типа коллекции и связанная с ней задача обхода (*итерирования*) созданной коллекции.

Однако для реализации пользовательского типа коллекции программисту приходится реализовывать множество вспомогательных конструкций, таких как реализация интерфейсов коллекций.

Еще одна важная особенность современных приложений – необходимость работы с большими объемами данных, что приводит к концепции поэлементной генерации коллекции «на лету».

Возможность прервать создание коллекции, и восстановить выполнение метода в том же самом месте существенно увеличивает гибкость и позволяет значительно упростить архитектуру приложения и повысить читаемость кода.

В современных языках программирования высокого уровня, таких как *C#* и *Python*, для реализации данных возможностей используется оператор *yield*.

На данный момент, для реализации возможности прервать генерацию элементов коллекции, и восстановить ее выполнение в том же месте, где она была прервана, программисту на языке PascalABC.NET необходимо реализовать громоздкий и трудный для быстрого понимания и изменения класс конечного автомата, управляющего состоянием выполнением генерации элементов коллекции.

В рамках данной работы рассматривается проектирование и реализация вспомогательной архитектуры, необходимой для реализации данного оператора в языке программирования PascalABC.NET, и ее встраивание в существующую инфраструктуру проекта.

Постановка задачи

Реализовать в языке программирования высокого уровня PascalABC.NET оператор **yield**, который позволит описывать методы-итераторы.

Использование **yield** для определения итератора исключает необходимость применения явного дополнительного класса (в котором содержится состояние перечисления) при реализации интерфейсов IEnumerable и IEnumerator для пользовательского типа коллекции.

Следующие подзадачи были поставлены для решения в рамках данного проекта:

1. Реализация оператора yield как *синтаксического сахара*
2. Развертка управляющих конструкций языка (*lowering*) для обеспечения возможности дальнейшего восстановления состояния метода-итератора (в котором использована конструкция yield) и продолжения его выполнения в любом месте.
3. Использование концепции паттерна *Visitor* для модификации уже сгенерированного синтаксического дерева.
4. Использование уровня синтаксиса везде, где это возможно; в случае невозможности – использовать так называемую “*вязкую семантику*”, т.е. вынос отдельных элементов “*сахарной*” конструкции на уровень семантики.

Исследование предметной области

Для реализации поставленных задач было выполнено исследование инфраструктуры проекта PascalABC.NET.

Компиляция программ проходит по общей схеме:

1. Построение AST
2. Создание кода по AST
3. Сборка

В зависимости от компилятора, возможно разделение этапов построения AST на построение *синтаксического* и *семантического* дерева, либо же маркировка синтаксического дерева атрибутами, необходимыми для дальнейшей генерации кода.

В проекте PascalABC.NET выбрана двухуровневая модель – сначала по тексту программы строится синтаксическое дерево, затем оно переводится в семантическое дерево, по которому выполняется генерация кода.

Процесс компиляции программы из исходного текста разделяется на следующие шаги:

1. Построение синтаксического дерева программы
2. Построение семантического дерева программы
3. Генерация IL-кода платформы .NET
4. Оптимизация
5. Сборка

Этапу оптимизации не уделяется значительного внимания, поскольку генерируется IL код, который затем оптимизируется JIT-компилятором платформы .NET.

Используя рефлексор .NET кода *ILSpy*, было проведено исследование структуры скомпилированной программы с использованием оператора *yield*.

Для реализации оператора *yield* дополнительно была изучена инфраструктура проекта *Roslyn* – платформы .NET компилятора от *Microsoft* с открытым исходным кодом.

Так же, как и в проекте PascalABC.NET, в Roslyn выбрана двухуровневая модель создания деревьев – синтаксического и семантического.

Развертка (*lowering*) – это преобразование конструкций, содержащих вложенные конструкции, в линейный список конструкций.

Преимуществом развернутого (*lowered*) кода над неразвернутым является возможность создания оператора *goto* к внутренним конструкциям развернутой конструкции.

В проекте Roslyn для реализации конструкции *yield* после построения *семантического* дерева перед генерацией кода выполняются его преобразования

1. Выполняется развертка (*lowering*) управляющих конструкций языка, таких как *for*, *while*, *foreach*, *if*.
2. После этого в семантическом дереве не остается циклических конструкций. Это позволяет выполнить переход (*goto*) к любому месту в теле метода.
3. Выполняется генерация семантических поддеревьев для лямбда-функций и *async/await* методов и замена их узлов на эти поддеревья. При этом выполняется создание вспомогательных типов для реализации этих конструкций языка и их добавление в семантическое дерево.
4. Выполняется генерация семантических поддеревьев (новых тел) для методов-итераторов (методов, содержащих конструкцию *yield*). Аналогично в семантическое дерево добавляются вспомогательные классы конечных автомата, управляющих выполнением методов-итераторов.
5. По полученному окончательному преобразованному семантическому дереву выполняется генерация кода и дальнейшая сборка модулей.

В силу удобства и большей легкости работы на уровне синтаксиса, было принято решение выполнять развертку (*lowering*) не на семантическом, а на синтаксическом уровне.

Для проведения развертки (*lowering*) были выбраны наиболее часто встречающиеся управляющие конструкции: *if, for, while, repeat, foreach, case*.

Алгоритмы Lowering

Развертка конструкции If

Псевдокод конструкции *If* выглядит следующим образом (возможны два варианта)

1. Сокращенный

```
if (condition)  
consequence
```

2. Полный

```
if (condition)  
consequence  
else  
alternative
```

Соответственно, для каждого варианта выполнен свой вариант развертки:

1. Сокращенный

```
GotoIfFalse condition afterif;  
consequence;  
afterif:
```

2. Полный

```
GotoIfFalse condition alt;  
consequence;  
goto afterif;  
alt:  
alternative;  
afterif:
```


Развертка конструкции **while**

Псевдокод конструкции *While* выглядит следующим образом:

```
while (condition)
    body;
```

Конструкция разворачивается в:

```
goto continue;
start:
{
body;
    continue:
    GotoIfTrue condition start;
}
break:
```

Развертка конструкции **Repeat**

Псевдокод конструкции *Repeat* выглядит следующим образом:

```
repeat
body;
until condition;
```

Конструкция разворачивается в:

```
goto continue;
start:
{
    continue:
        body;
    GotoIfFalse condition continue;
}
break:
```

Развертка конструкции For

Псевдокод конструкции *For* выглядит следующим образом:

```
for initializer direction limit do
    body
```

Конструкция разворачивается в:

```
initializer;
goto end;
start:
body;
continue:
increment;
end:
GotoIfTrue condition start;
break:
```

Развертка конструкции Foreach

Псевдокод конструкции *Foreach* выглядит следующим образом:

```
foreach iterator in collection do
    body
```

Конструкция разворачивается в:

```
enumerator = collection.GetEnumerator()
while (enumerator.MoveNext())
{
    current = enumerator.Current;
    body;
}
```

Затем выполняется *lowering* полученного кода.

Развертка конструкции Case

Псевдокод конструкции *Case* выглядит следующим образом:

```
case value of
  caseVariant1: statement1;
  caseVariant2: statement2;
  ..
  caseVariantN: statementN;
else: statementElse;
```

Конструкция разворачивается в:

```
if value satisfy caseVariant1 then statement1
else if value satisfy caseVariant2 then state-
ment2
..
else if value satisfy caseVariantN then state-
mentN
else statementElse
```

Затем выполняется lowering полученного кода.

Основные концепции реализации метода-итератора как синтаксического сахара

Проблема сохранения состояния метода-итератора

Основной сложностью реализации метода-итератора является сохранение его состояния между вызовами. Это означает, что значения всех имён, используемых в методе, должны сохраняться между операторами `yield`.

Однако следует уточнить, что некоторые имена, такие как описанные вне метода-итератора и содержащего его класса, могут быть изменены между вызовами метода-итератора, и следует использовать их актуальные значения.

Для решения этой проблемы тело метода-итератора преобразовывается в детерминированный конечный автомат по специальному алгоритму, который будет рассмотрен далее.

Архитектура метода-итератора как синтаксического сахара

Была спроектирована следующая вспомогательная архитектура, основной целью которой являлось обеспечить сохранение состояния метода-итератора и возможность итерирования создаваемой им коллекции.

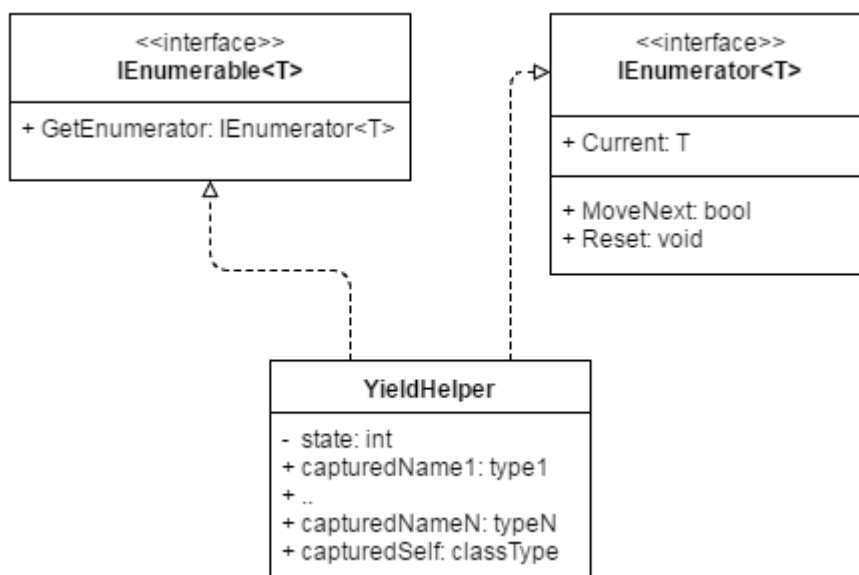


Рис 1. Вспомогательная архитектура сахарной реализации метода-итератора

Основной идеей является создание вспомогательного класса, которому делегируется управление выполнением метода-итератора и сохранение его состояния между вызовами.

Вспомогательный класс содержит состояние выполнения метода-итератора (как детерминированного конечного автомата) и реализует интерфейсы `IEnumerable` и `IEnumerator` для контролирования итерирования по создаваемой коллекции.

Тело метода-итератора заменяется на новое, в котором содержится создание данного вспомогательного класса.

Оригинальное тело метода преобразуется в детерминированный конечный автомат, и переносится в метод `IEnumerator.MoveNext`.

В новом теле метода-итератора выполняется так называемый **захват имён**, смысл и содержание которого поясняется далее.

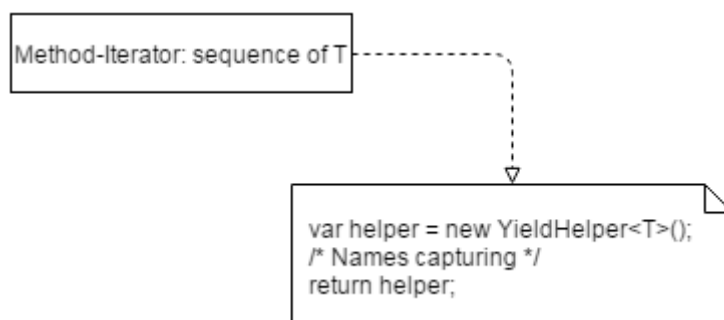


Рис 2. Архитектура сахарной реализации метода-итератора

Захват имён

Проблема сохранения состояния метода-итератора между его вызовами приводит к необходимости *захвата имён*, используемых в этом методе.

Это означает, что необходимо восстановить контекст выполнения метода (значения локальных переменных, значения фактических параметров метода) ровно в том виде, в каком он находился на момент вызова предыдущего оператора `yield`.

Для этого необходимо обеспечить их сохранение во вспомогательном классе, который управляет выполнением метода итератора.

Формально имена в методе можно разбить на несколько категорий (классов):

1. Локальные переменные
2. Формальные параметры метода
3. Имена, принадлежащие классу, содержащему метод-итератор
4. Глобальные имена (все остальные)

Сложность и алгоритм захвата имени зависит от категории, к которой оно принадлежит. Также, метод-итератор может быть описан вне класса как глобальная функция, что позволяет не выполнять захват имен третьей категории.

Классификация имён

Для того чтобы выполнить захват имени, необходимо отнести его к одной из четырех категорий, описанных выше. Информацию о формальных параметрах метода-итератора можно получить из синтаксического узла, в котором содержится его описание.

Для сбора информации о локальных переменных и об именах, принадлежащих классу, содержащему метод-итератор, используются легковесные вспомогательные визиторы. Собранная информация передается из них в визитор, выполняющий захват имен в методе-итераторе. Имена, не попавшие ни в одну из этих категорий, по умолчанию относятся к глобальным (в том случае, если метод-итератор описан как глобальный, т.е. вне класса). Следует заметить, что ес-

ли метод-итератор описан в некотором классе, то разрешение принадлежности имени этому классу/глобальным именам делегируется этапу семантики.

Алгоритмы захвата имён

Захват локальных переменных метода-итератора

Простейший алгоритм захвата локальных переменных состоит из следующих шагов:

1. Найти все объявления локальных переменных в методе-итераторе
2. Выполнить переименование (во избежание возможных конфликтов имен)
3. Удалить все объявления локальных переменных в методе-итераторе
4. Объявить новые имена локальных переменных публичными полями вспомогательного класса, управляющего методом-итератором

Но возникают следующие проблемы, связанные с особенностями языка:

1. Поскольку существует возможность присвоить значение переменной сразу при объявлении, опуская явное указание типа, тип переменной заранее точно неизвестен.

Например

```
var x := 666.6;
```

2. Существуют ограничения на значения, которые можно присвоить полю класса при его объявлении.

3. Вместе это приводит к невозможности описать поле класса без начального значения, не указывая его тип.

Поскольку точно узнать тип имени на этапе синтаксиса невозможно, решение проблемы – прибегнуть к “вязкой семантике”, делегируя определение типа имени этапу семантического анализа. Для этого создается вспомогательный синтаксический узел, который обрабатывается специальным образом на этапе перевода синтаксиса в семантику. Далее информация из этого узла используется другими синтаксическими узлами, которым она необходима.

В связи с особенностями перевода синтаксического дерева в семантическое и особенностями работы с контекстом компиляции, необходимо выполнить следующие дополнительные действия:

1. Создать копию метода-итератора.
2. Заменить в нем объявления локальных переменных на узлы-обертки.
3. Вставить вспомогательный метод в синтаксическое дерево.
4. На этапе семантики при обработке узлов-оберток получить типы содержащихся в них локальных переменных.
5. При объявлении имен локальных переменных как публичных полей класса подставить эти вычисленные на этапе семантики типы как типы полей.

Это приводит к следующему преобразованию алгоритма захвата локальных переменных:

1. Найти все объявления локальных переменных в методе-итераторе.
2. Выполнить переименование (во избежание возможных конфликтов имен).
3. В зависимости от наличия у объявления переменной начального значения
 - a. Если начальное значение указано – преобразовать объявление в присваивание имени значения.
 - b. Иначе – удалить объявление.
4. Для переменных, описанных в секции описания переменных метода-итератора и имеющих начальные значения, вставить присваивания именам их значений в пролог тела метода-итератора
5. Объявить имена локальных переменных публичными полями вспомогательного класса, управляющего методом-итератором.
 - a. Если тип переменной указан явно – использовать в поле класса этот тип
 - b. Иначе – использовать вычисленный на этапе семантики при помощи узла-обертки тип

Захват формальных параметров метода-итератора

Захват формальных параметров метода-итератора происходит по следующему алгоритму

1. Объявить имена формальных параметров метода-итератора полями вспомогательного класса, управляющего методом-итератором.
2. В новом теле метода-итератора выполнить присваивание этим полям фактических значений формальных параметров метода-итератора.

Захват имен класса, содержащего метод-итератор

Захват имен, принадлежащих классу, содержащему метод-итератор, выполняется по следующему алгоритму:

1. Объявить поле `<4>__self` типа класса, содержащего метод-итератор во вспомогательном классе, управляющем методом-итератором.
2. В новом теле метода-итератора присвоить этому полю `self` содержащего его класса.
3. Найти в теле метода-итератора все имена, принадлежащие классу, содержащему метод-итератор.
4. Заменить эти имена на обращение через захваченный `<4>__self` вида `<4>__self.имя`.

Но возникает следующая проблема: на этапе синтаксиса определить точно принадлежность имени классу, содержащему метод-итератор, невозможно. Например, в силу того, что имя может быть описано в его базовом классе, находящемся во внешней .NET-сборке.

Решение проблемы аналогично захвату локальных переменных - прибегнуть к “вязкой семантике”, делегируя определение принадлежности имени классу этапу семантического анализа.

Для этого также создается вспомогательный синтаксический узел-обертка, обрабатываемый на уровне перевода синтаксического дерева в семантическое.

Но, как следует из постановки задачи, необходимо минимизировать использование семантики. Поэтому замена узла имени на обертку происходит только в случае, если имя не найдено среди имен класса, содержащего метод-итератор.

Захват глобальных имен

В эту категорию попадают все имена, не относящиеся к другим 3 категориям (локальные переменные, формальные параметры и имена класса). Специального алгоритма захвата таких имен не предусмотрено.

Особенности реализации метода-итератора как синтаксического сахара

Особенности Lowering в PascalABC.NET

Схема lowering, рассмотренная в предыдущем разделе, не учитывает следующих особенностей управляющих конструкций в PascalABC.NET.

1. В циклах for и foreach возможно описание переменной цикла непосредственно в операторе цикла. Это приводит к тому, что имя переменной видимо только в мини-пространстве имен этого цикла. Однако схема lowering, описанная выше, предусматривает создание служебного имени вида `<>LV_name` в lowered-теле цикла для переменной цикла, объявленной как `var name` в операторе цикла. Как результат, следующий код, содержащий ошибку:

```
var i := 777;  
for var i := 666 to 777 do ... // повторное объявление имени i!
```

в ходе lowering преобразуется в код вида

```
var i := 777;  
var LV_i := 666;  
...  
lowered-for-code
```

И скомпилируется несмотря на наличие ошибки в исходном коде программы.

Для решения этой проблемы создан специальный визитор, проверяющий наличие подобных ошибок ДО выполнения lowering.

2. Возможны описания одинаковых имен в так называемых мини-пространствах имен в теле метода.

```

procedure MiniScopeDemo;
begin
    begin
        var x := 666;
    end;
    begin
        var x := 777;
    end;
end;

```

Поскольку после выполнения lowering вызывается визитор, удаляющий лишние begin..end (в том числе, созданные в ходе lowering), возможен конфликт имен.

Код

```

begin
    var x := 666; // мини-пространство имен 1
end;
begin
    var x := 777; // мини-пространство имен 2
end;

```

Будет преобразован в

```

begin
    var x := 666;
    var x := 777;
    // Переменная x уже объявлена в этом про-
    странстве имен
end;

```

Для решения этой проблемы используется визитор, выполняющий преобразование имен в мини-пространствах имен в уникальные.

Код

```
begin
    begin
        var x := 666; // мини-пространство имен 1
    end;
    begin
        var x := 777; // мини-пространство имен 2
    end;
end;
```

Будет преобразован в

```
begin
    begin
        var $x1 := 666; // мини-пространство имен 1
    end;
    begin
        var $x2 := 777; // мини-пространство имен 2
    end;
end;
```

В результате выполненного преобразования после удаления лишних `begin..end` конфликт имен не возникнет.

3. В операторе `case` пересечения диапазонов значений в разных ветках `case` являются ошибкой, однако после выполнения `lowering` это никак не проверяется.

Для решения этой проблемы было принято решение создать копию метода-итератора с новым именем, но идентичным телом и добавить ее ДО описания метода-итератора. В результате, при наличии ошибки, бэкенд компилятора ее обнаружит и не приступит к выполнению `lowering` в основном методе.

4. Так как вспомогательные методы для поиска ошибок и определения типов локальных переменных вставляются ДО описания метода-итератора, чтобы их обошел бэкенд компилятора ДО обхода метода-итератора, необходимо добавить предописание метода-итератора, если он описан вне класса, в секцию объявлений модуля.

```
function Gen: sequence of integer; forward;
```

Также следует учитывать возможное наличие такого предописания, которое может привести к ошибке повторного объявления.

Особенности реализации оператора `yield` в PascalABC.NET

Некоторые конструкции языка являются редко используемыми или реализация поддержки оператора `yield` в таких конструкциях имеет высокую сложность.

В промышленных компиляторах, таких как Roslyn от компании Microsoft, реализована поддержка совместимости `yield` и этих конструкций, но в данной работе было принято решение запретить их совмещение.

1. Использование оператора *`yield`* запрещено при наличии блока *`try..except..finally`*
2. Использование оператора *`yield`* запрещено при наличии оператора *`with`*
3. Использование оператора *`yield`* запрещено при наличии оператора *`lock`*
4. Использование оператора *`yield`* запрещено в *лямбда-выражениях*

Также к особенностям реализации оператора `yield` в PascalABC.NET является синтаксис возврата значения функции при помощи переменной ***Result***. Поэтому в методах-итераторах ее явное использование запрещено.

Алгоритм преобразования тела метода-итератора в детерминированный конечный автомат

1. Выполнить lowering тела метода
2. Найти все идентификаторы в теле метода и отклассифицировать их
3. Выполнить захват:
 - a. Локальные переменные поднять как поля вспомогательного класса и удалить их описания из тела
 - b. Формальные параметры поднять как поля вспомогательного класса
 - c. Имена, принадлежащие классу, обернуть через захваченный self класса.
 - d. Остальные имена оставить без изменений
4. На месте тела метода создать вспомогательный класс, реализующий интерфейс IEnumerable и добавить в него поля из шага (2)
5. Сформировать конечный автомат в методе MoveNext вспомогательного класса
 - a. Количество yield = количество состояний + начальное (0)
 - b. Сформировать секцию case по состояниям + обработка начального (0) состояния
 - c. Последовательно обходить операторы, помещая их в секцию case для обрабатываемого состояния
 - d. Если встречен оператор yield, заменить его на последовательность операторов:

```
current := <yielded-value>
state := <next-state>
return true
```
 - e. Если это не последний оператор, то сформировать новое состояние в case, иначе - закрыть case

- f. Если встречен оператор, помеченный меткой, то добавить в секцию после case метку и последовательность операторов до конца процедуры/до следующего yield. Если не в case встречен оператор, помеченный меткой, то его не обрабатываем, оставляем на том же месте.

Тестирование

В системе PascalABC.NET реализовано автоматическое тестирование по определенному набору тестов перед сборкой проекта и генерацией инсталляционных файлов, что обеспечивает оперативное обнаружение внесенных ошибок.

Тесты делятся на 2 категории:

1. Тесты компиляции.
2. Тесты выполнения.

Тесты компиляции считаются пройденными, если программа компилируется без ошибок.

Тесты выполнения должны содержать операторы `assert`, проверяющие правильность выполнения написанной программы.

Был разработан и интегрирован в систему тестовый набор из 10 тестов выполнения, 10 тестов компиляции, покрывающий базовые случаи возможной развертки оператора `yield`.

В данной работе приведены 4 теста выполнения и 4 теста компиляции как пример. Так как тесты выполнения частично покрывают тесты компиляции, большее внимание уделено негативным тестам, которые не компилируются.

Тестирование глобального метода-итератора без захвата переменных

```
function Gen: sequence of integer;
```

```
begin
```

```
    yield 777;
```

```
end;
```

```
begin
```

```
    assert (Gen.SequenceEqual (Seq(777)), 'Incorrect collection');
```

```
end.
```

Тестирование глобального метода-итератора с захватом локальной переменной

```
function Gen: sequence of integer;  
begin  
  var x := 777;  
  yield x;  
  x := 888;  
  yield x;  
end;  
  
begin  
  assert (Gen.SequenceEqual (Seq(777, 888)), 'Incorrect  
collection');  
end.
```

В приложении приведен код перевода данного метода-итератора в синтаксический сахар, полученный при помощи PrettyPrinter-визитора.

Узлы *yield_node* разворачиваются алгоритмом в изменение состояния детерминированного конечного автомата.

Непереведенными остаются узлы вспомогательные узлы-обертки для определения типов локальных переменных *yield_var_def_statement_with_unknown_type* и *yield_unknown_expression_type*, они переводятся на этапе перевода синтаксиса в семантику.

Тестирование метода-итератора шаблонного класса с захватом имени базового класса

```
type B<T> = class
  x: T;
end;
type A<T> = class (B<T>)
  function Gen: sequence of T;
  begin
    yield x;
  end;
end;
begin
  var aa := new A<real>();
  assert (aa.Gen.SequenceEqual (Seq(0.0)), 'Incorrect collection');
end.
```

Тестирование метода-итератора в классе с захватом имени из базового класса

```
type B = class
  x := 777;
end;

type A = class (B)
  function Gen: sequence of integer;
begin
  yield x;
end;

  procedure ChangeBaseField;
begin
  x := 888;
end;
end;

begin
  var aa := A.Create;

  assert (aa.Gen.SequenceEqual (Seq(777)), 'Incorrect collection');

  aa.ChangeBaseField;

  assert (aa.Gen.SequenceEqual (Seq(888)), 'Incorrect collection');
end.
```

Тестирование компиляции метода-итератора со статическим именем в теле

```
type A = class
class i: integer;
class function f: sequence of integer;
begin
  while i < 100 do
  begin
    Inc(i);
    yield i;
  end;
end;
end;
begin
  A.f.Println;
end.
```

Тестирование компиляции метода-итератора с использованием переменной result в теле (негативный тест)

```
function Gen: sequence of integer;
begin
  yield 777;
  result := Seq(1,2,3); // Запрещено использовать result
end;
begin
  assert(Gen.SequenceEqual(Seq(777)), 'Incorrect collection');
end.
```

Тестирование компиляции метода-итератора с вложенными подпрограммами (негативный тест)

```
// Вложенные подпрограммы запрещены использовать вместе с
yield
function f: sequence of integer;
    function ff: sequence of integer;
    begin
        yield 2;
    end;
begin
    yield 1;
end;

begin
end.
```

Тестирование компиляции метода-итератора с блоком try..except (негативный тест)

```
// Блок try..except запрещен в подпрограммах с yield
function Gen: sequence of integer;
begin
    try
        yield 777;
    except
    end;
end;
begin
end.
```

Приложения

Для тестирования “сахарной” развертки метода-итератора использовался специальный PrettyPrinter-визитор, позволяющий изучить структуру развернутого кода.

Код развертки глобального метода-итератора с захватом локальной переменной

```
function Gen: sequence of integer;
function <yield_helper_error_checkerr>Gen: sequence of integer;
begin
  var $x__0 := 777;
  yield $x__0
  $x__0 := 888
  yield $x__0
end
function <yield_helper_locals_type_detector>Gen: sequence of integer;
begin
  PascalABCCompiler.SyntaxTree.yield_var_def_statement_with_unknown_type
  $x__0 := 777;
  yield $x__0
  $x__0 := 888
  yield $x__0
end
type
  clyield#1 =
class (System.Collections.Generic.IEnumerable<integer>, System.Collections.Generic
.IEnumerator<integer>)
  public modifier
  <$x__0>MethodLocalVariable__1: PascalABCCompil-
er.SyntaxTree.yield_unknown_expression_type;
  yield_unknown_expression_type
  <>1__state: integer;
  <>2__current: integer;
  constructor ();
  begin
  end
  procedure Reset ();
  begin
  end
end
```

```

function MoveNext(): boolean;
begin
  case <>1__state of
    0:
      begin
        <$x__0>MethodLocalVariable__1 := 777
        <>2__current := <$x__0>MethodLocalVariable__1
        <>1__state := 1
        Result := True
        exit
      end
    1:
      begin
        <$x__0>MethodLocalVariable__1 := 888
        <>2__current := <$x__0>MethodLocalVariable__1
        <>1__state := 2
        Result := True
        exit
      end
    2:
      begin
        exit
      end
  end;
end
function System.Collections.IEnumerator.get_Current(): object;
begin
  Result := <>2__current
end
function System.Collections.IEnumerable.GetEnumerator(): System.Collections.IEnumerator;
begin
  Result := Self
end
function get_Current(): integer;
begin
  Result := <>2__current
end
function GetEnumerator(): System.Collections.Generic.IEnumerator<integer>;
begin
  Result := Self
end
procedure Dispose();
begin
end
end;
function Gen: sequence of integer;
begin
  var res := new clyield#1();
  Result := res
end
begin
  assert(Gen.SequenceEqual(Seq(777,888)), 'Incorrect collection')
end

```


Заключение

В результате данной работы был реализован оператор `yield` как синтаксически сахарная конструкция.

В ходе работы была выполнена развертка управляющих конструкций языка при помощи паттерна `Visitor`, модифицирующего уже сгенерированное синтаксическое дерево.

Разработан алгоритм преобразования тела метода-итератора в детерминированный конечный автомат.

Там, где уровня синтаксиса оказалось недостаточно, использовалась “вязкая семантика”, позволяющая получить информацию на уровне семантического анализа.

Работа над проектом велась в открытом репозитории на `GitHub`[6], было сделано 30 коммитов, изменения в затронутых уже существующих файлах (в основном, `syntax_tree_visitor.cs`) проекта были помечены псевдокомментариями для обеспечения удобства поиска изменений.

Также в ходе работы над проектом были исправлены некоторые ошибки `PrettyPrinter`-визитора в основной части проекта.

Список использованной литературы

1. [Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес](#). Приёмы объектно-ориентированного проектирования. Паттерны проектирования [Текст] / перевод с англ. – А. Слинкин. СПб.: Питер, 2013. – 368 с. – ISBN 978-5-459-01720-5 Фаулер М., Бек К., Брант Д., Робертс Д., Апдайк У. Рефакторинг: улучшение существующего кода — Спб: Символ-Плюс, 2009.
2. Фаулер М., Бек К., Брант Д., Робертс Д., Апдайк У. Рефакторинг: улучшение существующего кода — Спб: Символ-Плюс, 2009.
3. Taking a tour of Roslyn [Электронный ресурс] - Matt & the Managed Languages Team -
URL: <http://blogs.msdn.com/b/csharpfaq/archive/2014/04/03/taking-a-tour-of-roslyn.aspx> (дата обращения 14.06.2016)
4. NET Compiler Platform ("Roslyn") Overview. [Электронный ресурс] -
URL: <https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview> (дата обращения 14.06.2016)
5. Iterator block implementation details: auto-generated state machines [Электронный ресурс] –
URL:
<http://csharpindepth.com/Articles/Chapter6/IteratorBlockImplementation.aspx> (дата обращения 14.06.2016)
6. PascalABC.NET at GitHub [Электронный ресурс] –
URL: <https://github.com/pascalabcnet/pascalabcnet>