МИНОБРНАУКИ РОССИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Институт математики, механики и компьютерных наук
им. И. И. Воровича
Кафедра алгебры и дискретной математики

Саушкин Роман Сергеевич

РЕАЛИЗАЦИЯ ЗАМЫКАНИЙ В PASCALABC.NET

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

по специальности 010400 – Прикладная математика и информатика

Научный руководитель –
доц., к.ф.-м.н. Михалкович Станислав Станиславович

Рецензент –
доц., к.т.н. Демяненко Яна Михайловна

Ростов-на-Дону
2015

СОДЕРЖАНИЕ

Введение

Система программирования PascalABC.NET - это реализация языка Object Pascal, сочетающая простоту языка Паскаль и огромные возможности платформы .NET. Язык PascalABC.NET можно охарактеризовать как язык, поддерживающий самые современные средства языков программирования: обобщенные классы и подпрограммы, интерфейсы, перегрузка операций, исключения, сборка мусора, n-мерные динамические массивы и т.д.

PascalABC.NET работает на платформе .NET и генерирует исполнимые файлы под эту же платформу. Платформа .NET - программная платформа, выпущенная компанией Microsoft в 2002 году. Основой платформы является общеязыковая среда исполнения Common Language Runtime (CLR), которая подходит для разных языков программирования. Функциональные возможности CLR доступны в любых языках программирования, использующих эту среду. Хотя .NET официально рассчитана на работу под операционными системами семейства Microsoft Windows, существуют независимые проекты (прежде всего это Mono и Portable.NET), позволяющие запускать программы .NET на некоторых других операционных системах.

Данная работа посвящена реализации замыканий в PascalABC.NET. Замыкание — функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся её параметрами. Говоря другим языком, замыкание — функция, которая ссылается на свободные переменные в своем контексте. При использовании замыканий говорят, что функция захватывает переменные из внешнего контекста. Замыкания в PascalABC.NET представлены лямбда-выражениями. Первоначально лямбда-выражения не поддерживали захват переменных из контекста, внешнего по отношению контекста этого лямбда-выражения. Единственными видами переменных, которые можно было использовать внутри тела лямбда-выражений, являлись глобальные переменные и поля класса. Обуславливалось это тем, что при компиляции функция, генерируемая

для лямбда-выражения, добавлялась либо на глобальный уровень, либо как метод класса. В обоих случаях, таким образом, корректный доступ к «захватываемым» переменным обеспечивался как положительный побочный результат уже реализованных алгоритмов. Данная работа посвящена исправлению этого недостатка.

Диссертация состоит из четырех глав. В первой главе «Лямбда-выражения и современные языки программирования» представлены определения лямбда-выражения и основных понятий, связанных с ними. Также приведен анализ реализаций лямбда-выражений в современных языках программирования, таких как C# и C++. Во второй главе «Общая структура компилятора PascalABC.NET» описываются основные стадии компиляции в системе PascalABC.NET, а также внутренние представления программ в процессе компиляции. В третьей главе «Синтаксические средства лямбда-выражений в PascalABC.NET» представлены основные синтаксические правила языка PascalABC.NET, позволяющие работать с лямбда-выражениями. В четвертой, последней и самой объемной главе приводится описание алгоритмов реализации захвата переменных в лямбда-выражениях, а также представлены исчерпывающие примеры для более точного понимания работы этих алгоритмов. Кроме того, четвертая глава содержит раздел «Вывод параметров типов при инстанцировании шаблонных подпрограмм при реализации лямбда-выражений». Несмотря на то, что основная тема, которой посвящена данная работа, является реализация замыканий, автором была проделана некоторая работа по модификации алгоритма вывода типов в шаблонных подпрограммах, так как информация о типах является неотъемлемой частью при обработке замыканий. В этом разделе приводится описание модификации этого алгоритма в случае, когда в шаблонные подпрограммы в качестве фактических параметров передаются лямбда-выражения, тип параметров и возвращаемого значения которых не были указаны явно и которые необходимо вывести автоматически.

Постановка задачи

1. Реализовать замыкания в PascalABC.NET с поддержкой следующих видов захвата переменных:
    - захват локальных переменных;
    - захват параметров процедур и функций;
    - захват переменных циклов `for` и `foreach`;
    - захват полей класса, включая поля, не являющиеся публичными.
2. Обеспечить поддержку одновременного использования нескольких видов захвата.
3. Реализовать поддержку лямбда-выражений внутри шаблонных подпрограмм.
4. Модифицировать алгоритм вывода параметров типов при инстанцировании шаблонных подпрограмм при реализации лямбда-выражений.

Глава 1. Лямбда-выражения и современные языки программирования

1.1 Определение лямбда-выражения

Как уже было сказано ранее, PascalABC.NET, являясь компилятором под платформу .NET, открывает возможности работы с современными конструкциями объектно-ориентированных языков. Поддержка делегатов является одним из таких средств.

Делегат – это тип, который представляет собой ссылки на методы с определенным списком параметров и возвращаемым типом. При создании экземпляра делегата этот экземпляр можно связать с любым методом с совместимой сигнатурой и возвращаемым типом. Метод можно вызвать (активировать) с помощью экземпляра делегата.

Делегаты используются для передачи методов в качестве аргументов к другим методам. До введения в PascalABC.NET поддержки лямбда-выражений именованные методы были единственным способом объявления делегатов. Так, например, для передачи подпрограммы в качестве аргумента другой подпрограммы нужно было сначала полностью описать эти передаваемые подпрограммы и лишь после этого использовать их имена при вызове другой подпрограммы (см. пример 1).

```
function f1(x: integer): boolean;
begin
   result := x mod 2 = 0;
end;

function f2(x: integer): integer;
begin
  result := x;
end;

begin
  Seq(3,5,2,3,5,7,0,3,1,1,2,6,3).Where(f1).OrderBy(f2).Print();
end.
```

Пример 1.

Лямбда-выражение — это специальный синтаксис для объявления анонимных подпрограмм по месту их использования. С помощью лямбда-выражений можно создавать локальные функции, которые затем можно передавать в другие функции в качестве аргументов или возвращать из них в качестве значения.

В примере 2 показано, как изменится программа из примера 1 с использованием лямбда-выражений. Из примера видно, насколько удобно использование лямбда-выражений при передаче в качестве аргумента в подпрограмму.

```
begin
  Seq(3,5,2,3,5,7,0,3,1,1,2,6,3)
     .Where(x -> x mod 2 = 0).OrderBy(x -> x).Print();
end.
```

Пример 2.

Наиболее подробно и полно средства, посвященные поддержке лямбда-выражений в PascalABC.NET, а также особенности использования лямбда-выражений будут описаны в следующих разделах.

1.2 Основные понятия, связанные с лямбда-выражениями

В данном разделе будут рассмотрены основные понятия, специфичные для лямбда-выражений, отличающие их от привычных в императивных языках процедур и функций.

1.2.1 Вывод типов в лямбда-выражениях

В общем случае при описании любой процедуры или функции в языках программирования требуется полностью указывать сигнатуру той или иной подпрограммы, включающую в себя информацию о типах формальных

параметров этой подпрограммы, а в случае функции также типа возвращаемого значения этой подпрограммы.

В лямбда-выражениях, реализуемых современными языками программирования, обычно используется иной подход: при использовании лямбда-выражения в большинстве случаев не требуется указывать типов формальных параметров и типа возвращаемого значения этого лямбда-выражения. Компилятор в таком случае способен определить эту информацию самостоятельно, основываясь на типах выражений, составляющих тело данного лямбда-выражения, а также на типах переменных внешнего окружения.

Вывод типов возможен в двух случаях:

- при присваивании лямбда-выражения переменной;
- при передаче лямбда-выражения в качестве аргумента другой подпрограммы.

В первом случае указывать типы параметров лямбда-выражения излишне, так как вся необходимая информация о типах содержится в типе переменной, которой присваивается данное лямбда-выражение. Во втором случае нет необходимости явно указывать типы потому, что информация об этих типах берется из типа формального параметра подпрограммы, на место которого передается лямбда-выражение.

Средства современных языков программирования по автоматическому выводу типов способствуют более продуктивной работе с данным языком программирования, позволяя не заботиться о правильном написании типов всех выражений, используемых в тексте программы. Кроме того, программы, написанные на таких языках программирования, получаются лаконичными и легко читаемыми, что облегчает дальнейшую поддержку кода.

В данной работе будет рассмотрена реализация алгоритма вывода типов параметров и возвращаемого значения в лямбда-выражениях в среде программирования PascalABC.NET при их передаче в качестве аргументов в

подпрограммы при поиске верного из нескольких перегруженных вариантов этих подпрограмм, а также будут рассмотрены особенности алгоритма вывода дженерик-параметров при инстанцировании с использованием лямбда-выражений.

1.2.2 Область действия переменных в лямбда-выражениях

При написании программ для решения необходимых задач программист описывает набор локальных переменных, глобальных переменных, подпрограммы с собственным набором локальных параметров и т.д. Каждая из описанных переменных и параметров имеют собственную область видимости, которая определяет, откуда к ней можно будет обратиться. Например, на глобальные переменные программы можно ссылаться из любой другой части программы, в то время как к формальному параметру процедуры можно обращаться только из данной процедуры. При исполнении программы по мере того как завершается выполнение блока, в котором определена та или иная переменная, из памяти удаляются все локальные переменные, определенные внутри данного блока, так что к этим переменным теряется всякий доступ, и запись и чтение значения этих переменных становятся невозможными.

В случае лямбда-выражений доступ к переменным, определенным в некотором блоке, выполнение которого уже закончилось, в то время как лямбда-выражение все еще продолжает существовать, становится возможным. Большинство современных языков программирования поддерживают лямбда-выражения, способные ссылаться на внешние переменные, находящиеся в области видимости, содержащую данное лямбда-выражение. Например, в языке C# лямбда-выражения могут ссылаться на переменные, находящиеся в области видимости метода, в котором данное лямбда-выражение определено, либо в области видимости типа, которое содержит это лямбда-выражение.

Переменные, полученные таким способом, сохраняются для использования в лямбда-выражениях, даже если бы в ином случае они оказались за границами области действия.

В данной работе будет описана реализация алгоритма и последовательность действий, выполняемых для обеспечения поддержки захвата различных видов переменных лямбда-выражениями в системе PascalABC.NET.

1.3 Лямбда-выражения в некоторых языках программирования

Лямбда-выражения первоначально нашли применение в функциональном программировании. Функциональное программирование – это парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании). Функциональное программирование противопоставляется парадигме императивного программирования, которая описывает процесс вычислений как последовательное изменение состояний.

В последние годы поддержка функциональных возможностей нашло широкое распространение в императивных языках программирования, таких как C++, C# и т.д. К таковым можно отнести и поддержку лямбда-выражений. В данном разделе рассмотрим средства поддержки лямбда-выражений в C++ и C#. Это позволит нам сравнить и проанализировать, как различные императивные языки могут существовать совместно с функциональными возможностями, и впоследствии сравнить особенности реализации лямбда-выражений этих языков с реализацией лямбда-выражений в PascalABC.NET.

1.3.1 Лямбда-выражения в C++

В языке программирования C++ лямбда-выражение – это краткая форма записи анонимного функтора. В C++ «функтор» является сокращением от «функциональный объект». Функциональный объект является экземпляром класса C++, в котором определён operator(). Если определить operator() для класса C++, то мы получим объект, который действует как функция, но может также хранить состояние.

В C++ синтаксическая конструкция, представляющая лямбда-выражение, преобразуется компилятором в функтор, после чего создается экземпляр этого функтора и подставляется на место этого лямбда-выражения. Проиллюстрируем все вышесказанное на примере (см. пример 3).

```
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <vector>
using namespace std;
void main()
{
    vector<int> arr;
    for (int i = 0; i < 10; i++)
    {
        arr.push_back(i);
    }
    int result =
        count_if(arr.begin(), arr.end(), [] (int _n)
        {
            return (_n % 2) == 0;
        });
    cout << result << endl;
}
```

Пример 3(а).

```
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <vector>
using namespace std;
```

Пример 3(б).

```
class lambda1
```

```
{
    public: bool operator ()(int _x) const {return (_x % 2)==0;}
};
void main()
{
    vector<int> arr;
    for (int i = 0; i < 10; i++)
    {
        arr.push_back(i);
    }
    int result = count_if(arr.begin(), arr.end(), lambda1());
    cout << result << endl;
}
```

Пример 3(б, окончание).

В части (а) примера 3 приведена программа, в которой в качестве предиката в функцию `count_if` передается лямбда-выражение, которое определяет, является ли переданное целое число четным. В части (б) примера 3 показано, что для данного лямбда-выражения компилятор генерирует функтор `lambda1`, и помещает тело лямбда-выражения в тело перегруженного оператора `operator ()`.

Лямбда-выражение в C++ всегда начинается с [ ], где [ ] может быть не пустым, а задавать так называемый список захвата, значение которого будет объяснено позже. Затем идет необязательный список параметров и, наконец, тело лямбда-выражения. Явное указание типов параметров является обязательным, в отличие от типа возвращаемого значения, которое компилятор способен определить самостоятельно, основываясь на типе выражения, которое возвращает тело лямбда-выражения. В некоторых сложных случаях, например, когда в теле лямбда-выражения используется несколько операторов `return`, и каждый из них присутствует в операторе `if` или `switch`, компилятор не может вычислить тип возвращаемого значения автоматически, и тогда необходимо указывать тип явно в виде `-> bool`, где вместо `bool` может стоять любой требуемый тип. Для примера 3 полная

синтаксическая конструкция лямбда-выражения тогда будет иметь вид, указанный в примере 4.

```
[] (int _n) -> bool
{
    return (_n % 2) == 0;
}
```

Пример 4.

Список захвата (capture list), о котором упоминалось ранее, служит для указания всех переменных, описанных в контексте, внешнем по отношению к лямбда-выражению, и захватываемых этим лямбда-выражением. Указывать такой список необходимо для того, чтобы компилятор имел информацию о том, что внутри тела лямбда-выражения существуют ссылки на переменные из области видимости, содержащей это лямбда-выражение, и смог сгенерировать скрытые поля в классе генерируемого функтора для хранения состояния этих переменных и впоследствии обеспечить доступ к значениям этих переменных из `operator ()`, соответствующего данному лямбда-выражению. Все сказанное иллюстрирует пример 5. Как и в примере 3, в части (а) примера приведен код программы, в части (б) содержится примерный код, генерируемый компилятором (директивы `#include` опущены для экономии места).

```
void main()
{
    vector<int> arr;
    int val = 5;
    for (int i = 0; i < 100; i++)
    {
        arr.push_back(i);
    }
    int result =
        count_if(arr.begin(), arr.end(), [val] (int _n)
        {
            return _n > val;
        });
    cout << result << endl;
}
```

Пример 5(а).

```
class lambda2
{
    private: int val;
    public:
          lambda2(int _val): val(_val) {}
          bool operator ()(int _x) const {return _x > val;}
};

void main()
{
    vector<int> arr;
    int val = 5;
    for (int i = 0; i < 100; i++)
    {
        arr.push_back(i);
    }
     int result = count_if(arr.begin(), arr.end(), lambda2(val));
     cout << result << endl;
}
```

Пример 5(б).

Как видно из примеров 3 и 5, компилятор C++ перегружает `operator ()` как `const`. Это значит, что захваченные из внешнего контекста переменные нельзя будет изменять внутри тела лямбда-выражения. Для того чтобы обойти это ограничение, можно при использовании лямбда-выражения указать для него ключевое слово `mutable`, которое должно следовать непосредственно после списка формальных параметров лямбда-выражения.

В C++ разрешены два типа захвата переменных, или, как они еще называются, замыканий:

- по значению;
- по ссылке.

В первом случае изменения захватываемых лямбда-выражением переменных внутри этого лямбда-выражения не отражаются на самой переменной, несмотря на то, что указано слово `mutable`. Во втором случае эти изменения отражаются на захватываемой переменной.

Язык C++ позволяет указывать способ захвата переменных как на уровне отдельной переменной, так и для всех переменных сразу. По умолчанию используется захват по значению. Можно также не указывать каждую переменную в списке захвата по отдельности: вместо этого можно просто указать способ захвата по умолчанию, и тогда все переменные из внешнего контекста, которые используются внутри лямбда-выражения, будут захвачены компилятором автоматически. В таблице 1 приведены некоторые примеры определения типа замыканий.

Определение типа замыканий	**Семантический смысл типа замыканий**
[]	без захвата переменных из внешней области видимости
[=]	все переменные захватываются по значению
[&]	все переменные захватываются по ссылке
[x, y]	захват x и y по значению
[&x, &y]	захват x и y по ссылке
[in, &out]	захват in по значению, a out — по ссылке
[=, &out1, &out2]	захват всех переменных по значению, кроме out1 и out2, которые захватываются по ссылке
[&, x, &y]	захват всех переменных по ссылке, кроме x

Таблица 1. Некоторые примеры определения типа замыканий в C++

1.3.2 Лямбда-выражения в C#

Язык C# предоставляет больше свободы при написании кода с использованием лямбда-выражений, нежели язык C++. Например, язык C#

обязывает явно указывать типы формальных параметров и типа возвращаемого значения лямбда-выражений только в тех случаях, когда автоматически это сделать действительно сложно либо невозможно. В остальных ситуациях при написании лямбда-выражений обычно не требуется указывать тип входных параметров, поскольку компилятор может выводить этот тип на основе тела лямбда-выражения, типа делегата параметра и других факторов, как описано в спецификации языка C#.

При выводе типов компилятор C# руководствуется следующими основными правилами:

- лямбда-выражение должно содержать то же число параметров, что и тип делегата, которому присваивается или на место которого подставляется;
- каждый входной параметр в лямбда-выражении должен быть неявно преобразуемым в соответствующий параметр делегата.
- возвращаемое значение лямбда-выражения (если таковое имеется) должно быть неявно преобразуемым в возвращаемый тип делегата.

Аналогично языку C++, язык C# позволяет строить лямбда-выражения, которые могут ссылаться на внешние переменные, находящиеся в области метода, в котором определена лямбда-функция, или в области типа, который содержит лямбда-выражение.

В отличие от языка C++, в котором разрешены замыкания по значению и по ссылке, язык C# реализует замыкания исключительно по ссылке. При этом не требуется указывать список захвата – компилятор сам определяет, какие переменные из внешнего контекста были захвачены внутри лямбда-выражений, и генерирует соответствующий вспомогательный код для их поддержки.

Еще одним отличием реализации лямбда-выражений в языке C# от C++ заключается в том, что в C# для каждого лямбда-выражения генерируется

метод с той же сигнатурой, что и исходное лямбда-выражение (с учетом выведенных типов), и ссылка на него подставляется вместо этого лямбда-выражения. В случае захвата переменных из внешнего контекста может генерироваться от одного до нескольких вспомогательных классов. Их вид зависит от того, одна или несколько переменных захватываются, учитываются разновидность этой переменной (локальная переменная, формальный параметр метода, поле класса), а также контекст, в котором эта переменная описана.

В основе реализации замыканий в PascalABC.NET, являющейся основной целью, поставленной при выполнении данной работы, во многом лежит идеология реализации замыканий в языке C#. Детали реализации замыканий в PascalABC.NET в различных ситуациях будут даны в следующих главах. Здесь же рассмотрим пример кода, генерируемого компилятором C# при замыкании лямбда-выражением локальной переменной метода (см. пример 6).

Как обычно, в части (а) примера приведен код программы, в части (б) содержится примерный код, генерируемый компилятором.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            var l = new List<int> {1, 2, 3, 4, 5, 2, 5, 6, 3};
            var v = 2;
            Console.WriteLine(l.Count(x => x > v));
        }
    }
}
```

Пример 6(а).

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace ConsoleApplication
{
    class Program
    {
        private sealed class <>c__DisplayClass2
        {
            public int32 v;
            public bool <Main>b__1(int32 x)
            {
                return (x > this.v);
            }
        }
        private static void Main(string[] args)
        {
            <>c__DisplayClass2 CS$<>8__locals3 =
                    new <>c__DisplayClass2();
            List<int> l =
                    new List<int> {1, 2, 3, 4, 5, 2, 5, 6, 3};
            CS$<>8__locals3.v = 2;
            Console.WriteLine(System.Linq.Enumerable.Count(l,
               new Func<int, bool>(CS$<>8__locals3.<Main>b__1)));
        }
    }
}
```

Пример 6(б).

Рассмотрим пример 6 подробнее. Как видно из примера, в лямбда-выражении имеется ссылка на локальную переменную `v`, определенную в вышележащем по отношению к этому лямбда-выражению блоке. Чтобы обеспечить корректность данного замыкания, компилятор сгенерировал класс `<>c__DisplayClass2`, в который поместил публичное поле `int32 v`, которое представляет захватываемую переменную. Далее, в месте, где создается и инициализируется переменная `v`, он создал оператор описания экземпляра класса `<>c__DisplayClass2`. После этого компилятор произвел замену всех обращений к переменной `v` на `CS$<>8__locals3.v`. При этом полю `v` было присвоено значение, которым инициализировалась переменная `v` в исходной

программе. Наконец, компилятор заменил лямбда-выражение на обращение к сгенерированному методу `<Main>b__1`, который представляет исходное лямбда-выражение и которое компилятор поместил в класс `<>c__DisplayClass2`. В результате этих преобразований создается код, который с одной стороны позволяет лямбда-выражению изменять значение захваченной переменной, а с другой стороны изменять значение захваченной переменной в той области видимости, в которой эта переменная была описана. К тому же, если бы мы захотели передать полученное лямбда-выражение как результат метода `Main`, то эта захваченная переменная, которая сейчас на самом деле является полем класса `<>c__DisplayClass2`, продолжила бы «жить» и за пределами метода `Main`, поскольку на нее есть ссылка из этого экземпляра сгенерированного класса, что не дает сборщику мусора произвести удаление этой переменной из памяти.

1.3.3 Запросы LINQ и лямбда-выражения

PascalABC.NET работает на платформе .NET и генерирует исполнимые файлы под эту же платформу. Это означает, что в программах, разрабатываемых в PascalABC.NET, открываются широкие возможности использования стандартной библиотеки классов .NET.

LINQ – это набор появившихся в .NET Framework 3.5 функций, которые значительно расширяют возможности синтаксиса языков для платформы .NET. «LINQ to Objects» является подмножеством LINQ и означает использование запросов LINQ с любой коллекцией `IEnumerable` или `IEnumerable<T>`. В общем смысле LINQ to Objects представляет собой новый подход к коллекциям. Традиционная обработка коллекций предполагает написание сложных циклов `for` и `foreach`, определяющие порядок извлечения данных из коллекции. При использовании LINQ пишется декларативный код, описывающий, какие данные необходимо извлечь.

Обычно методы LINQ для определения данных, которые необходимо извлечь из коллекции, принимают в качестве аргумента тип делегата.

До появления в PascalABC.NET лямбда-выражений приходилось описывать функцию, которую необходимо передать в запрос LINQ, отдельно и передавать ее имя. Так как такие функции зачастую нужны один раз, то количество таких функций в программе растет пропорционально количеству вызовов методов LINQ. Такой стиль написания программ сильно загромождает код, делая его трудночитаемым.

Так как лямбда-выражения – это анонимные функции, с помощью которых можно создавать типы делегата, то их можно использовать для передачи в качестве фактических параметров в запросы LINQ. Появление лямбда-выражений в PascalABC.NET с поддержкой автоматического вывода типов параметров и возвращаемого значения, включая случай использования лямбда-выражений внутри дженерик-методов, а также замыканий переменных из внешнего контекста значительно упростило написание запросов LINQ, одновременно расширяя возможности языка.

Ранее в примерах 1 и 2 раздела 1.1 было проиллюстрировано использование запросов LINQ `Where` и `OrderBy` без использования лямбда-выражений, а также с их использованием. Приведем эти примеры также здесь (см. примеры 7 и 8).

```
function f1(x: integer): boolean;
begin
   result := x mod 2 = 0;
end;
function f2(x: integer): integer;
begin
  result := x;
end;

begin
  Seq(3,5,2,3,5,7,0,3,1,1,2,6,3).Where(f1).OrderBy(f2).Print();
end.
```

Пример 7.

```
begin
  Seq(3,5,2,3,5,7,0,3,1,1,2,6,3)
```

```
    .Where(x -> x mod 2 = 0).OrderBy(x -> x).Print();
end.
```

Пример 8.

Глава 2. Общая структура компилятора PascalABC.NET

В данной главе будет в общих чертах описан процесс компиляции программ в системе PascalABC.NET.

Рис. 1. Стадии компиляции в PascalABC.NET

На рис. 1 изображена последовательность выполнения стадий компиляции программ в системе PascalABC.NET. Вначале исходный текст программы поступает на вход синтаксическому анализатору, именуемому также парсером, и разбирается им в так называемое синтаксическое дерево. Синтаксическое дерево содержит представленный в виде дерева текст программы. В процессе разбора текст программы проверяется лишь на соответствие синтаксическим конструкциям, семантика на этом уровне практически не учитывается. Фактически, синтаксическое дерево – это текст программы, переведенный парсером в удобное для последующей обработки представление. Синтаксическое дерево не содержит никакой информации о сущностях, определенных в программе. Например, на синтаксическом уровне

нет таблицы символов, содержащей данных обо всех переменных, описанных в программе, и их типах. Поэтому в процессе синтаксического анализа невозможно производить такие сложные действия, как вывод типов формальных параметров и возвращаемого значения лямбда–выражения или анализ захваченных из внешнего контекста в лямбда-выражениях переменных.

После того как синтаксическое дерево построено, оно переводится в семантическое дерево с помощью визитора (конвертера деревьев). Семантическое дерево – это внутреннее представление программы, содержащее исчерпывающую информацию о правильной программе. Оно учитывает все правила, которые сформулированы в описании языка. В семантическом дереве, например, хранится такая информация, как тип переменных, пространство имен, к которому эти переменные относятся, для подпрограмм проверяется соответствие количества и типов формальных и фактических параметров и т.д. Алгоритмы поиска имен в таблице символов, алгоритмы выбора перегруженных подпрограмм и многие другие алгоритмы, необходимые для анализа семантики, работают на этом этапе. Получаемое на выходе семантическое дерево содержит всю информацию, необходимую для генерации IL-кода.

Наконец, по семантическому дереву генерируется IL -код с помощью визитора по семантическому дереву.

Глава 3. Синтаксические средства лямбда-выражений в PascalABC.NET

Синтаксис лямбда-выражений в языке PascalABC.NET имеет общие черты с синтаксисом лямбда-выражений в языке C#. В то же время, в отдельных случаях в синтаксических правилах лямбда-выражений в PascalABC.NET прослеживается сходство с синтаксисом традиционных подпрограмм (процедур и функций) языка Паскаль.

Ниже с помощью расширенной формы Бэкуса–Наура приведен общий вид лямбда–выражений, которые способен распознать парсер PascalABC.NET (рис. 2).

```
<лямбда–выражение> ::=
     <параметр>  –>  <тело лямбда–выражения >  |
     [function | procedure] ({<список параметров[: тип]>} [ : тип])
     –>  <тело лямбда–выражения >
<список параметров> ::=
     <параметр> | <список параметров>, <параметр>
<тело лямбда–выражения > ::=
     <выражение> | begin <список операторов> end
```

Рис. 2. Общий вид лямбда-выражений в PascalABC.NET.

Как видно из рис. 2, каждое лямбда-выражение начинается со списка формальных параметров, которые задаются слева от лямбда-оператора –>. Список параметров может отсутствовать. В свою очередь, формальные параметры могут быть описаны как с явным указанием их типов, так и без них. В случае, когда тип формальных параметров опущен, в процессе компиляции будет произведена попытка их вывода.

Стоит отметить одно существенное отличие лямбда-выражений от обычных подпрограмм языка PascalABC.NET. В отличие от последнего,

формальные параметры лямбда-выражений не могут быть описаны с ключевыми словами `var` и `const`. Иными словами, параметры лямбда-выражений всегда передаются по значению, а не по ссылке.

После списка формальных параметров указывается тип возвращаемого значения лямбда-выражения. Как и в случае с типами формальных параметров, тип возвращаемого значения можно опускать. В этом случае в процессе компиляции будет произведена попытка его вывода. Вывод типа возвращаемого значения основывается на типе делегата, лежащего в основе лямбда-выражения, а также непосредственно на теле этого лямбда-выражения.

Справа от лямбда-оператора `->` помещается тело лямбда-выражения, которое может состоять из одного оператора или блока операторов. В последнем случае блок операторов заключается в операторные скобки `begin...end`, подобно тому, как это делается в случае обычных подпрограмм PascalABC.NET.

В PascalABC.NET можно выделить два типа лямбда-выражений: лямбда-функция и лямбда-процедура. В первом случае, для того чтобы компилятор смог корректно вывести тип возвращаемого значения лямбда-выражения (если он явно не указан), в теле этого лямбда-выражения необходимо использовать операторы вида `result := <выражение>`.

Ниже приведем несколько примеров использования лямбда-выражений (примеры 9, 10).

```
begin
  var a: Func<integer, integer>:=function (x: integer) -> x * x;
  writeln(a(5));
end.
```

Пример 9.

```
function SquaredFunction(f: Func<real, real>; x: real): real;
begin
  result := f(x) * f(x);
end;
begin
  writeln(SquaredFunction(x -> sin(x) * cos(x) + tan(x), 45));
end.
```

Пример 10.

Глава 4. Семантический анализ лямбда-выражений

В данной главе будут описаны алгоритмы обработки лямбда-выражений на этапе семантического анализа компилятора PascalABC.NET. Мы рассмотрим изменения алгоритма вывода параметров типов при инстанцировании дженерик-подпрограмм при передаче в них в качестве фактических параметров лямбда-выражений; средства анализа переменных, захватываемых в лямбда-выражениях из внешнего по отношению к ним контекста, а также алгоритмы, предназначенные для осуществления захвата этих переменных.

4.1 Класс WalkingVisitor

Прежде чем перейти к обсуждению алгоритмов, перечисленных выше, рассмотрим класс, являющийся базовым для многих обходчиков (визиторов) синтаксических деревьев, который лег в основу многих классов, участвующих в реализации алгоритмов.

Класс `WalkingVisitor` является потомком класса `AbstractVisitor`, который, в свою очередь, является реализацией интерфейса `IVisitor`. В интерфейсе `IVisitor` представлены методы со следующей сигнатурой:

```
void visit(<тип_узла_синтаксического_дерева>).
```

На каждый тип узла синтаксического дерева представлен отдельный метод `visit`.

В классе `AbstractVisitor` представлена реализация методов интерфейса `IVisitor`. Все методы в этом классе объявлены виртуальными и имеют пустое тело.

В классе `WalkingVisitor` эти методы переопределены, и здесь уже представлена полная базовая реализация обхода всех типов узлов синтаксического дерева. Для всех методов продолжает сохраняться цепочка виртуальности, поэтому в случае необходимости при создании нового

визитора можно лишь переопределить способ обхода нужных узлов, оставив реализацию обхода остальных узлов из `WalkingVisitor`. Именно на таком подходе основаны многие классы, используемые при реализации алгоритмов, перечисленных в начале главы. Обход в визиторе начинается в методе `ProcessNode`.

Стоит отметить еще одну важную полезную деталь класса `WalkingVisitor` . В нем имеется два делегата `OnEnter` и `OnLeave`, которые вызываются непосредственно соответственно перед обходом конкретного узла и непосредственно после него. Это позволяет производить дополнительные действия, которые нужно выполнить в моменты до и после анализа узла. Данная возможность также широко использовалась автором при выполнении данной работы.

На рис. 3 изображена диаграмма классов, описанных в данном разделе.

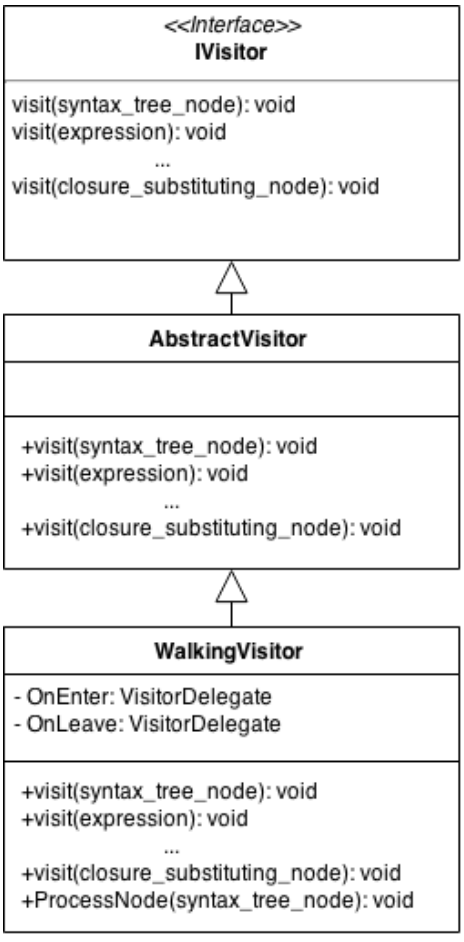

Рис. 3. Диаграмма классов визиторов.

4.2 Стадии компиляции лямбда-выражений

В процессе семантического анализа лямбда-выражения проходят через ряд стадий компиляции. Можно выделить четыре основные фазы:

- стадия вывода типов. На этом этапе происходит восстановление (или, иными словами, автоматический вывод) типов формальных параметров и типа возвращаемого значения лямбда-выражений. После того как данная фаза успешно завершена, синтаксические узлы, представляющие собой типы параметров и возвращаемого значения, содержат в себе корректное определение семантических типов, как простых, так и инстанций дженериков. Данная стадия выполняется первой, так как для последующей обработки необходима исчерпывающая информация о типах.
- стадия обработки захваченных переменных. В ходе выполнения этой фазы происходит анализ переменных, захватываемых лямбда-выражениями, их типов, контекста, в котором они были описаны; генерируются вспомогательные синтаксические структуры, поиск всех обращений к захваченным переменным и т.д. Более подробно эти процессы будут рассматриваться далее.
- стадия семантического анализа структур, сгенерированных на стадии обработки захваченных переменных. Структуры, полученные на предыдущей фазе, являются синтаксическими узлами, представляющие классы в PascalABC.NET. Именно эти классы и анализируются на данном этапе.
- финальная стадия. К этому моменту уже произведены все необходимые действия по выводу типов и захвату переменных, и на данном этапе происходит обход основного анализируемого блока программы с учетом всех изменений, произведенных на предыдущих этапах: визитором обходятся все сгенерированные методы, соответствующие лямбда-выражениям, не захватившим

ни одной переменной из внешнего контекста, и поэтому не подпадающие под критерии обработки на этапах 2 и 3; также обходятся все узлы, соответствующие заменам обращений к захваченным переменным, и т.д.

Стадии обработки лямбды-выражений представлены перечислением, приведенным на рис. 4. Элементы перечисления представлены в том же порядке, в каком идут в описании выше. Этап `None` соответствует тому, что основной цикл обработки серии этапов обработки лямбда-выражений еще не начат.

```
public enum LambdaProcessingState
{
    None,
    TypeInferencePhase,
    ClosuresProcessingPhase,
    ClosuresProcessingVisitGeneratedClassesPhase,
    FinishPhase
}
```

Рис. 4. Стадии компиляции лямбда-выражений.

Указанный цикл этапов обработки лямбда-выражений происходит при семантическом анализе синтаксического узла `SyntaxTree.block`. Данный синтаксический узел может представлять как основной блок программы, так и блок подпрограмм. При семантическом анализе этого узла сначала происходит его обход с целью обработки лямбда-выражений, то есть для него запускается выполнение стадий 1-3 из вышеуказанной последовательности стадий, и затем окончательный его обход на стадии 4.

4.3 Вывод параметров типов при инстанцировании шаблонных подпрограмм при реализации лямбда-выражений

В 2008 году в системе программирования PascalABC.NET была введена поддержка шаблонов функций. В число семантических средств анализа шаблонов функций входит алгоритм выведения инстанции шаблона подпрограммы по списку фактических параметров. Выведение позволяет не указывать явно, какую инстанцию шаблонной подпрограммы следует вызвать – решение о выборе инстанции примет компилятор.

В связи с введением в PascalABC.NET лямбда-выражений необходимо было модифицировать алгоритм выведения инстанции шаблона, расширив возможности анализа этим алгоритмом выводимых параметров типов с учетом типов формальных параметров и типа возвращаемого значения лямбда-выражений, которые, в свою очередь, являются фактическими параметрами, подставляемыми в данный шаблон.

Алгоритм выведения инстанций шаблонов, представленный методами `DeduceFunction` и `DeduceInstanceTypes`, описан в [3]. Ниже приведены модификации, внесенные автором данной работы в этот алгоритм. Изменения в алгоритме выделены ***жирным курсивом на сером фоне*** (см. рис. 5 и 6).

```
function_node DeduceFunction(function_node func,
                             expressions_list fact,
                              List<SyntaxTree.expression>
                              syntax_nodes_parameters = null)
1. lambda_syntax_nodes :=
     ВыбратьФактическиеПараметры-Лямбды(syntax_nodes_parameters)
   lambda_in_parameters := lambda_syntax_nodes.Количество > 0
2. saved_lambdas_states :=
     СохранитьСостоянияЛямбд(lambda_syntax_nodes)
3. Если fact_count < formal_count то
     Если formal[fact_count].ЗначениеПоУмолчанию = null и
          не formal[fact_count].ЯвляетсяParamsПараметом то
       вернуть null;
```

Рис. 5. Модификации метода `DeduceFunction`.

```
   Иначе Если fact_count > formal_count то
```

```
    Если formal[formal_count-1] .ЯвляетсяParamsПараметом то
      Цикл от i:=formal_count-1 до fact_count-1 делать
        Если не DeduceInstanceTypes(
            formal[formal_count-1].ТипПараметра.ТипЭлементов,
            fact[i].ТипПараметра, deduced, nils) то
          вернуть ложь;
      count_params_to_see := formal_count - 1;
    Иначе
      вернуть null;
4. need_params_work := count_params_to_see > 0 и
      formal[count_params_to_see].ЯвляетсяParamsПараметром;
5. Если need_params_work то
     count_params_to_see--;
6. contine_trying_to_infer_types = true
   formal_delegates = null;
7. Пока continue_trying_to_infer_types делать
     previous_deduce_state :=
        СохранитьПредыдущееСостояниеВыведенныхПараметровТипов

     Цикл от i:=0 до count_params_to_see-1 делать
       Если не DeduceInstanceTypes(
           formal[i].ТипПараметра,
           fact[i].ТипПараметра, deduced, nils) то
             ВосстановитьСостоянияЛямбд(lambda_syntax_nodes,
                                        saved_lambdas_states)
             вернуть ложь;
     Если lambda_in_parameters то
       Если formal_delegates == null то
         formal_delegates :=
           ВыделитьИзформальныхПараметровСоответствующиеЛямбдам
       Для каждого formal_delegate из formal_delegates делать
         lambda_syntax_node :=
           lambda_syntax_nodes[formal_delegate.Имя]
         Если не TryToDeduceTypesInLambda(lambda_syntax_node,
                         formal_delegate.ФормальныйДелегат,
                         deduced, nils,
                         out on_lambda_body_compile_exception) то
           ВосстановитьСостоянияЛямбд(lambda_syntax_nodes,
                                      saved_lambdas_states)
           Если on_lambda_body_compile_exception != null то
             БроситьИсключение
             FailedWhileTryingToCompile
             LambdaBodyWithGiven
             ParametersException
               (on_lambda_body_compile_exception)
             вернуть ложь;
```

Рис. 5. Модификации метода `DeduceFunction` (продолжение).

```
    current_deduce_state :=
```

```
        СохранитьТекущееСостояниеВыведенныхПараметровТипов
    continue_trying_to_infer_types := СостоянияРазличаются
                                      (previous_deduce_state,
                                       current_deduce_state)
8. ВосстановитьСостоянияЛямбд(lambda_syntax_nodes,
                                      saved_lambdas_states)
9. Если need_params_work то
     tmp_deduced := deduced.СоздатьКопию;
     tmp_nils := nils.СоздатьКопию;
     Если не DeduceInstanceTypes(
         formal[count_params_to_see].ТипПараметра,
         fact[count_params_to_see].ТипПараметра,
         deduced, nils) то
       deduced := tmp_deduced;
       nils := tmp_nils;
       Если не Если не DeduceInstanceTypes(
          formal[count_params_to_see].ТипПараметра.ТипЭлементов,
          fact[count_params_to_see].ТипПараметра,
          deduced, nils) то
         вернуть ложь;
10. Цикл от i:=0 до func_generic_parameters_count-1 делать
     Если deduced[i] = null то
       вернуть ложь;
11. Цикл от i:=0 до nils.ЧислоЭлементов-1 делать
     Если не ТипСовместимCNil(deduced[nils[i]]) то
       вернуть ложь;
12. вернуть func.СоздатьИнстанцию(deduced);
```

Рис. 5. Модификации метода `DeduceFunction` (окончание).

```
bool DeduceInstanceTypes(type_node formal_type, type_node
fact_type, type_node[] deduced, List<int> nils)
1. Если formal_type = fact_type
     вернуть истину;
2. Если ТипыНеявноПриводимы(fact_type, formal_type)
     вернуть истину;
3. Если formal_type.ЯвляетсяШаблоннымПараметромПодпрограммы то
 3.1. Если fact_type = УзелДляТипаNIL то
        nils.Добавить(formal_type.НомерШаблонногоПараметра);
        вернуть истину;
 3.2. Если deduced[formal_type.НомерШаблонногоПараметра]=null то
         deduced[fact_type.НомерШаблонногоПараметра]:=fact_type;
         вернуть истину;
```

Рис. 6. Модификации метода `DeduceInstanceTypes`.

```
 3.3. Если deduced[formal_type.НомерШаблонногоПараметра] ≠
      fact_type то
        вернуть ложь;
 3.4. вернуть истину;

4.                              ...

5. Если formal_type.ЯвляетсяДелегатом то
     Если fact_type.ЯвляетсяЛямбдаВыражением то
       ВывестиПараметрыТиповИзВыведенныхТиповПараметровЛямбды
       Если ВывестиУдалось то
         вернуть истину;
       вернуть ложь;

6.                              ...
```

Рис. 6. Модификации метода `DeduceInstanceTypes` (окончание).

Опишем теперь изменения алгоритма, приведенные выше на псевдокоде, более подробно.

Алгоритм выведения параметров типов шаблона при использовании лямбда-выражений является итерационным и основан на сравнении состояний о выведенных типах на предыдущей и текущей итерациях.

Метод `DeduceFunction` на вход принимает список синтаксических узлов, представляющие собой фактические параметры, передаваемые в шаблонную подпрограмму. Этот список необходим для того, чтобы точно дифференцировать фактические параметры-лямбды от остальных фактических параметров. Список параметров-лямбд сохраняется один раз в начале выполнения алгоритма в переменной `lambda_syntax_nodes` и используется далее при попытке выведения шаблонных параметров типов.

В переменной `saved_lambda_states` сохраняется информация о типах формальных параметров и типе возвращаемого значения лямбда-выражений до выведения шаблонных параметров типов. Это необходимо в случае нескольких перегруженных версий шаблонной подпрограммы. Дело в том, что при попытке выведения шаблонных параметров типов подпрограммы в

формальные параметры лямбда-выражений подставляются типы, выведенные на данный момент. Эти подставленные типы используются далее для вычисления реального типа возвращаемого значения данного лямбда-выражения. Если имеется несколько перегруженных версий шаблонной подпрограммы, то при выведении параметров типов из очередной версии этой подпрограммы нужно руководствоваться исключительно первоначальной информацией о типах параметров лямбда-выражений, являющихся фактическими параметрами данной шаблонной подпрограммы. Для этих целей и производится сохранение состояния типов параметров фактических параметров-лямбд.

На шагах 6 и 7 алгоритма происходит основная работа по выведению параметров типов шаблонной подпрограммы. Здесь запускается цикл, зависящий от переменной `continue_trying_to_infer_types`. Эта переменная говорит о том, нужно ли дальше пытаться выводить типы, либо уже достигнуто состояние, в котором типы выведены, либо можно с точностью сказать, что типы вывести невозможно.

Выполнения цикла начинается с того, что определяется состояние о выведенных на данный момент типах. Состояние представляется обычным массивом, хранящим индексы элементов массива `deduced`, в которых, в свою очередь, хранятся уже выведенные типы.

На следующем этапе производится попытка выведения параметров типов шаблона в методе `DeduceInstanceTypes` на основании имеющейся информации о типах. На этом шаге (см. рис. 6) анализируются уже известные типы параметров лямбда-выражений и сопоставляются шаблонным параметрам типов. Если такое выведение не возможно либо противоречиво, то из метода возвращается ложь. Иначе возвращается истина, и выполнение `DeduceFunction` продолжается.

Далее производится попытка подстановки выведенных типов в типы формальных параметров с целью вычисления типа возвращаемого значения

данного лямбда-выражения. Эти действия производятся в методе `TryToDeduceTypesInLambda`. В этом методе после подстановки типов в синтаксический узел, соответствующий данному лямбда-выражению, производится обход тела этого лямбда-выражения и вычисляется тип возвращаемого значения. Здесь стоит отметить, что если изначально существует несколько перегруженных версий шаблонной подпрограммы и на этом этапе выбрана неверная версия, то в процессе обхода тела лямбда-выражения ввиду подстановки неверных типов формальных параметров этого лямбда-выражения могут возникнуть ошибки компиляции. Если такая ситуация возникла, то экземпляр исключения конкретной ошибки будет возвращен из метода `TryToDeduceTypesInLambda`. При этом метод проанализирует, следует ли выбрасывать это исключение на верхний уровень и в случае необходимости делает это. На внешнем уровне это исключение сохраняется, текущая перегруженная версия шаблонной подпрограммы признается неверной, и анализируются остальные версии. В конечном итоге возможны две ситуации: либо найдена нужная версия подпрограммы (в этом случае создается ее инстанция), либо ни одна версия не является корректной с точки зрения списка фактических параметров. В этом случае последнее сохраненное исключение будет преобразовано в диагностическое сообщение, которое впоследствии будет показано пользователю.

В конце шага 7 метода `DeduceFunction` снова вычисляется состояние выведенных типов и сравнивается с предыдущим. Если это состояние не отличается от того, что было перед выполнением тела цикла шага 7, то можно сделать вывод о том, что на последующих итерациях ситуация не изменится, и продолжать попытки вывода типов следует прекратить.

Отметим, что приведенный алгоритм завершается ввиду ограниченности перегруженных версий подпрограмм и анализа состояний.

4.4 Реализация замыканий переменных из внешнего контекста в лямбда-выражениях

В данном разделе будут описаны детали реализации захвата переменных из внешнего контекста в лямбда-выражениях. Вначале будет дано описание стадий семантического анализа лямбда-выражений в процессе обработки замыканий. Далее каждый из этапов анализа будет рассмотрен более подробно. Наконец, будут приведены примеры, иллюстрирующие различные виды замыканий (захват локальных переменных, захват параметров подпрограмм, захват переменной циклов `for` и `foreach`, захват полей класса).

4.4.1 Стадии семантического анализа лямбда-выражений в процессе обработки замыканий

В процессе реализации замыканий в языке PascalABC.NET автором данной работы был разработан механизм, включающий в себя несколько стадий семантического анализа лямбда-выражений. Всего можно выделить три этапа, на каждом из которых подготавливаются данные для выполнения следующего этапа:

- сбор информации о захватываемых переменных, определение и запоминание контекста, в котором та или иная переменная описана. Здесь учитывается случай, когда в одном и том же лямбда-выражении захватывается несколько переменных, описанных на различных уровнях (глобальный уровень, локальный уровень, параметр процедуры и т.д.). Также учитывается ситуация, когда одна и та же переменная захватывается в нескольких лямбда-выражениях, каждое из которых может быть использовано в различных вложенных блоках и смежных блоках;
- анализ полученной на предыдущем этапе информации о захваченных переменных и генерация на основании этой информации синтаксических структур, представляющие собой классы и методы, соответствующие захваченным переменным и

лямбда-выражениям. Эти структуры используются для поддержания «жизни» захваченных переменных, что проявляется в корректной обработке и сохранении значений захваченных переменных и ссылочной целостности;

- обход полученных на предыдущем этапе синтаксических структур и замена всех ссылок на захваченные переменные и лямбда-выражения на обращения к полям и методам этих структур.

На рис. 7 представлена схема выполнения данного цикла стадий. Смысл всех использованных в схеме классов будет объяснен в дальнейших разделах.

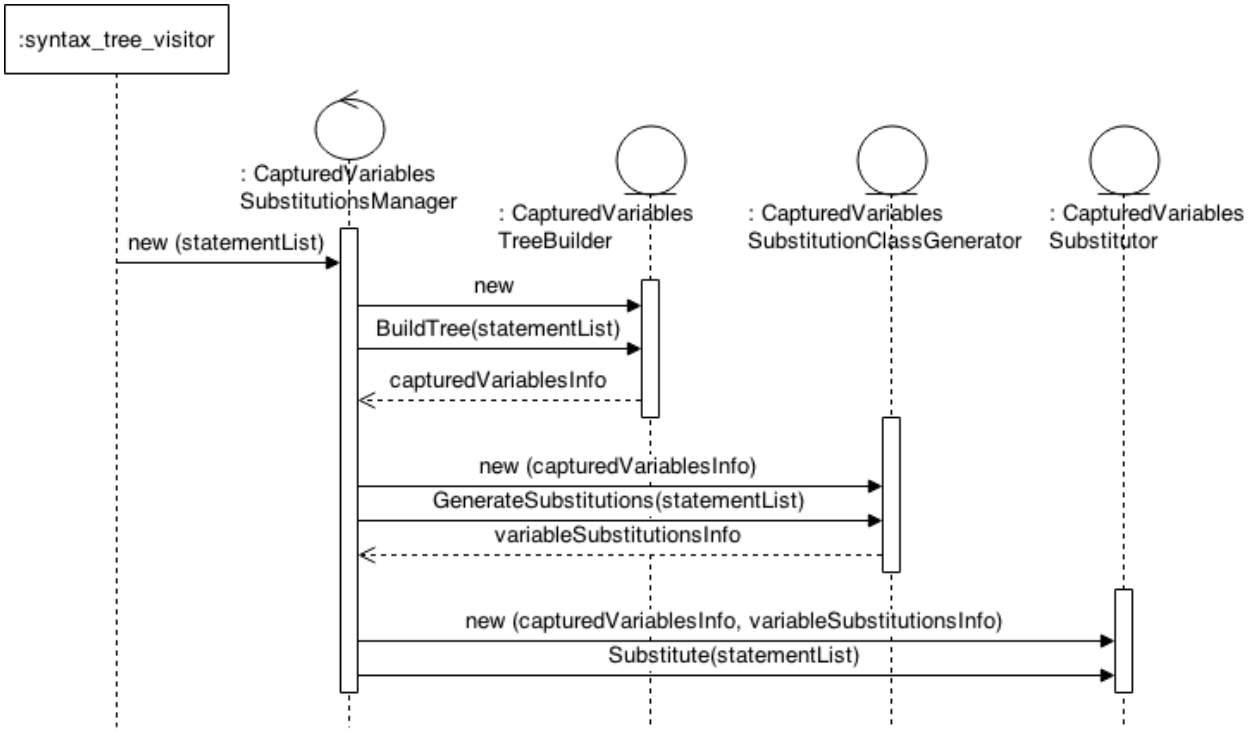

Рис. 7. Стадии семантического анализа лямбда-выражений при обработке замыканий

4.4.2 Анализ и определение захватываемых переменных

Для корректной обработки захватываемых переменных необходима следующая информация о них:

- имя переменной;
- тип переменной;
- контекст, в котором переменная описана;

- информация о лямбда-выражениях, которые захватывают переменную.

В системе программирования PascalABC.NET существует класс `SymbolInfo`, хранящий данные, перечисленные в первых трех пунктах выше. Это класс, относящийся к семантическим средствам анализа. Так как лямбда-выражения, не являясь отдельными сущностями в полученной сборке на IL-коде, в конечном итоге переводятся в узел семантического дерева, представляющий либо подпрограмму на глобальном уровне, либо метод класса, то единственным способом хранения информации о лямбда-выражениях, захватывающих ту или иную переменную, является список синтаксических узлов, представляющие эти лямбда-выражения. Кроме того, необходимо знать синтаксический узел, в котором та или иная переменная описана. Подобная информация нужна на этапе замены ссылок на захваченные переменные.

Грамматика языка PascalABC.NET такова, что при создании программ позволяется определять блоки операторов, ограниченные операторными скобками `begin...end`, которые, в свою очередь, могут содержать в себе вложенные блоки. В каждом из этих блоков можно описывать локальные переменные. Эти переменные впоследствии могут захватываться внутри лямбда-выражений, которые также могут быть описаны на разных уровнях-блоках.

С учетом аспектов, указанных выше, необходимо было разработать такую структуру, которая, во-первых, хранила бы всю необходимую информацию о захватываемых переменных, и, во-вторых, эта информация была бы структурирована в соответствии с иерархией вложенности блоков и других пространств имен, таких, как процедуры, функции и классы.

В связи с этим автором данной работы была разработана древовидная структура, узлами которой являются объекты класса `CapturedVariablesTreeNode`. Этот класс является абстрактным базовым

классом для набора других классов, каждый из которых соответствует одному из следующих пространств имен:

- блок операторов, ограниченных `begin...end`;
- класс;
- подпрограмма (процедура или функция);
- заголовок цикла `for`;
- заголовок цикла `foreach`;
- лямбда-выражение.

Каждый узел дерева хранит ссылку на родительский узел, список ссылок на дочерние узлы, список объектов типа `CapturedSymbolInfo`, содержащих информацию о переменных, описанных в данном пространстве имен; эта информация удовлетворяет требованиям, приведенным в начале главы; ссылку на узел синтаксического дерева исходной программы, соответствующий данному узлу описываемой древовидной структуры, и, наконец, список лямбда-выражений, описанных в данном пространстве имен. Автор выбрал подход создания отдельного класса для каждого типа пространств имен потому, что это позволяет на последующих стадиях семантического анализа дифференцировать различные случаи захвата переменных: является ли переменная локальной, либо она – формальный параметр подпрограммы или поле класса.

В пункте 4.1 данной работы был рассмотрен класс `WalkingVisitor`, который является обходчиком синтаксических деревьев. Для построения древовидной структуры, содержащей информацию о захватываемых переменных, был реализован визитор `CapturedVariablesTreeBuilder`, являющийся наследником `WalkingVisitor`. Ранее было сказано, что в случае необходимости при создании нового визитора можно переопределить способ обхода нужных узлов, оставив реализацию обхода остальных узлов из `WalkingVisitor`. Именно такой подход и был применен при реализации

CapturedVariablesTreeBuilder. Способ обхода в этом визиторе был изменен для следующих узлов синтаксического дерева:

- assign;
- dot_node;
- foreach_stmt;
- for_node;
- function_lambda_definition;
- ident;
- new_expr;
- statement_list;
- var_def_statement.

На вход визитору подается узел синтаксического дерева statement_list - список операторов, в котором необходимо обработать захват переменных лямбда-выражениями. В начале обхода создается корень древовидной структуры, и запускается обход одного за другим узлов-операторов.

Здесь следует отметить, что для удобства поиска все узлы древовидной структуры заносятся в словарь

```
Dictionary<int, CapturedVariablesTreeNode>
 _scopesCapturedVarsNodesDictionary;
```

Ключом этого словаря является целое число. Смысл этого числа следующий. В процессе семантического анализа каждое пространство имен в PascalABC.NET связывается с экземпляром класса Scope, который используется при работе с таблицей символов и который содержит данные о переменных, описанных в этом пространстве имен. Каждый такой экземпляр имеет уникальный порядковый номер – целое число. Именно это число и используется в качестве ключа словаря. Значением элементов словаря являются узлы древовидной структуры.

В ходе обхода синтаксического поддерева statement_list при анализе узлов типа var_def_statement создаются экземпляры класса

CapturedSymbolInfo, в которые заносится информация, однозначно идентифицирующая переменные, описанные в этом var_def_statement. Эта информация содержит данные, перечисленные в начале данного раздела. Далее вновь созданные экземпляры класса CapturedSymbolInfo добавляются в текущий CapturedVariablesTreeNode. Таким образом, собирается информация обо всех переменных, описанных в данном пространстве имен, причем эта информация представлена в виде, удобном для дальнейшей обработки.

Следует отметить, что при обходе заголовков циклов for и foreach, если в них создается переменная цикла, ее также необходимо добавить в список всех переменных, описанных в текущем пространстве имен. Это делается в соответствующих методах обхода.

Обращение к переменной x может происходить двумя способами:

- непосредственное обращение к данной переменной: x := 1;
- обращение к переменной через разыменование, если эта переменная имеет тип записи или класса: x.a := 1.

В обоих случаях обращения ко всем переменным фиксируются в словаре Dictionary<SubstitutionKey, List<ident>> _identsReferences.

Поясним смысл типа SubstitutionKey. SubstitutionKey – это класс, выступающий в качестве ключа в словаре _identsReferences. В этом словаре собирается информация обо всех обращениях ко всем переменным, которые потенциально впоследствии могут подвергнуться заменам на обращения к структурам, генерируемых на следующем этапе анализа замыканий. SubstitutionKey представлен тремя полями:

- _variableName: string - имя переменной, к которой идет обращение;

- `_syntaxTreeNodeWhereVaribleIsDeclared: syntax_tree_node` – синтаксический узел, в котором описана эта переменная;
- `_syntaxTreeUnderSubstitution: syntax_tree_node` – синтаксический узел, соответствующий пространству имен, в котором происходит обращение к переменной.

В итоге в словаре `_identsReferences` будут сгруппированы все обращения ко всем переменным в соответствии с признаками, перечисленными выше.

Далее, если данное обращение происходит из пространства имен лямбда-выражения, этот факт фиксируется добавлением текущего лямбда-выражения к списку `ReferencingLambdas` экземпляра класса `CapturedVarsSymbolInfo`, соответствующего захватываемой переменной. В свою очередь в узел лямбда-выражения записывается информация о том, что это лямбда-выражение захватило конкретную переменную.

В результате обхода списка операторов из узла `statement_list` будет построено дерево, содержащее полную информацию обо всех переменных, объявленных в пределах этого узла, контексте, в котором они описаны, всех обращениях к этим переменным, местам в коде, из которых эти обращения происходят. Данная структура передается на следующий этап обработки, который будет описан в следующем разделе. Здесь приведем пример построения древовидной структуры для простой программы. На рис. 8 приведен исходный текст программы. На рис. 9. изображено дерево, построенное в соответствии с алгоритмом, приведенным выше.

```
procedure pr(a: integer);
begin
  var b: integer := 2;
  begin
    var c: integer := 3;
    begin
      var d: integer := 4;
      writeln(Seq(1,2,3,4).Select(x -> x + a + b + d));
```

```
      d += 1;
      a *= 2;
      writeln(d);
    end;
  end;
  begin
    writeln(Range(1, 10).Select(x -> x + a + b));
    writeln(a);
  end;
end;

begin
  pr(5);
end.
```

Рис. 8. Пример программы с замыканиями

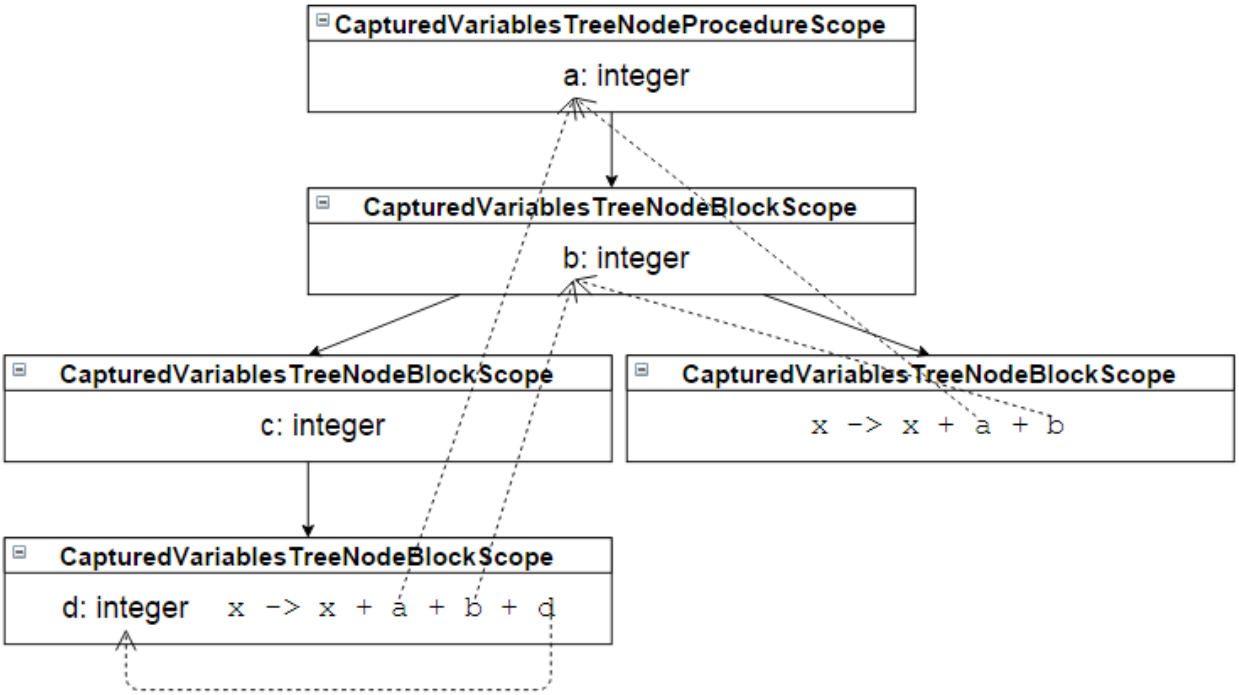

Рис.9. Дерево захваченных переменных

4.4.3 Построение вспомогательных синтаксических структур на основе данных, полученных в результате анализа захватываемых переменных

Древовидная структура, построенная на предыдущем этапе, является входными данными для следующего этапа. Цель второй стадии анализа состоит в том, чтобы по узлам этого дерева сгенерировать набор классов,

полями которых станут захваченные ранее переменные в лямбда-выражениях, а методами – тела этих лямбда-выражений.

Чтобы лучше понять смысл этих действий, рассмотрим следующую ситуацию. Пусть лямбда-выражение будет захватывать внешнюю переменную типа-значения, и время жизни этого лямбда-выражения будет превосходить время жизни переменных в стеке. Этого просто добиться путем возвращения делегата, ссылающегося на лямбда-выражение, либо сохранения этого делегата в качестве поля класса. Рассмотрим пример (см. пример 11):

```
function CreateAction(): System.Action;
begin
  var count := 0;
  result := procedure () -> begin
                              count += 1;
                              writeln('Count = ', count);
                            end;
end;

begin
  var action := CreateAction();
  action();
  action();
end.
```

Пример 11.

Результаты выполнения программы:

```
Count = 1
Count = 2
```

Может показаться, что переменная `count` располагается в стеке функции `CreateAction`, но вызов делегата будет осуществляться после завершения выполнения этого метода, из чего следует, что `count` будет уничтожена раньше, чем выполнится делегат. На самом же деле в стеке функции `CreateAction` переменной `count` нет. Вместо этого создается анонимный класс с открытым полем и создается метод, тело которого соответствует телу объявленного пользователем лямбда-выражения. Это способствует тому, что

внешне теперь кажется, что переменная `count` «живет» вне стека функции `CreateAction`.

В разделе 1.3 проводилось сравнение реализаций лямбда-выражений и захвата переменных из внешнего контекста в языках C++ и C#. При реализации лямбда-выражений в языке PascalABC.NET был заимствован подход языка C#, так как, по мнению автора данной работы, он выигрывает по сравнению с языком C++ по ряду причин. Во-первых, синтаксис лямбда-выражений выглядит более лаконично в C# в случае замыканий – не нужно указывать список захвата, как это делается в C++, компилятор сам определит захваченные переменные. Во-вторых, и C#, и PascalABC.NET являются компиляторами под платформу .NET, где не поддерживаются функциональные объекты, подобные C++. Поэтому представляется логичным при реализации лямбда-выражений в PascalABC.NET опираться на особенности платформы .NET, где для поддержки лямбда-выражений и замыканий генерируются вспомогательные классы и методы.

Рассмотрим теперь код, который будет генерироваться компилятором для примера 11. Код представлен (см. пример 12) на языке C#, так как получен в результате дизассемблирования сборки на IL-коде специальным инструментом.

В примерах далее код, генерируемый в результате компиляции лямбда-выражений, будет также приведен на языке C#. Специальные случаи захвата переменных, таких как переменные цикла и поля класса будут рассмотрены в разделе «4.4.5 Специальные случаи захвата переменных».

```
public class <>local_variables_class_0
{
    public int32 count;
    public void <>lambda1()
     {
          this.count = (this.count + 1);
          PABCSystem.PABCSystem.writeln
                    ("Count = ", (int)(this.count));
```

```
     }
}

public class Program
{
     public static Action CreateAction()
     {
          <>local_variables_class_0
                    <>local_variables_class_01 =
                    new Program132.<>local_variables_class_0();

          <>local_variables_class_01.count = 0;

          return
               new Action(<>local_variables_class_01.<>lambda1);
     }

     public static void Main()
     {
          Action action1 = Program.CreateAction();
          action1.Invoke();
          action1.Invoke();
     }
}
```

Пример 12.

Как видно из примера 12, компилятор генерирует класс `<>local_variables_class_0`, куда помещает поле `count` и метод `<>lambda_1`. Имена сгенерированного класса и метода имеют такой вид для того, чтобы обеспечить отсутствие конфликтов имен, поскольку пользователь самостоятельно не сможет создать класс или метод, имена которых будут содержать символы "<>". Также стоит отметить, что переменной `count` в стеке теперь нет. Вместо нее создается объект класса `<>local_variables_class_0` и используется его поле `count`. В результате переменная `count` значимого типа располагается не в стеке, а в управляемой куче, и не будет собрана сборщиком мусора до тех пор, пока будут оставаться ссылки на делегат, созданный функцией `CreateAction`.

Теперь, когда ясен смысл генерации вспомогательных структур, вернемся к рассмотрению второй стадии обработки замыканий.

Для генерации классов и множества синтаксических узлов, представляющие собой замены захваченных переменных был разработан класс `CapturedVariablesSubstitutionClassGenerator`. На вход ему подается древовидная структура, полученная на первом этапе анализа и которая была описана в предыдущем разделе. Далее в методе `VisitTreeAndBuildClassDefinitions` организуется рекурсивный обход по узлам этой древовидной структуры, в результате которого обрабатываются все переменные, описанные в рассматриваемом пространстве имен и захваченные хотя бы одним лямбда-выражением. Обработка каждой захваченной переменной ведется в методе `VisitCapturedVar`. На рис. 10 приведен общий алгоритм обработки захваченных переменных.

```
void VisitTreeAndBuildClassDefinitions
                          (CapturedVariablesTreeNode currentNode)

1. variablesFromThisScopeWhichWereCaptured :=
     ВыбратьВсеЗахваченныепеременныеИз(currentNode)

2. Если
     variablesFromThisScopeWhichWereCaptured.Количество > 0 то
       _capturedVarsClassDefs.СгенерироватьКласс
         ДляПространстваИменИДобавить
         ПоляЗахваченныеПеременные
         (currentNode, variablesFromThisScopeWhichWereCaptured)
       ДляВсех capturedVar из
          variablesFromThisScopeWhichWereCaptured делать
            VisitCapturedVar(currentNode, capturedVar)

3. ДляВсех childNode из currentNode.ДочернииеУзлы делать
      VisitTreeAndBuildClassDefinitions(childNode)
```

Рис. 10. Алгоритм обработки захваченных переменных

```
4. Если
     variablesFromThisScopeWhichWereCaptured.Количество > 0 то
       ДляВсех var из
         variablesFromThisScopeWhichWereCaptured делать
           ДляВсех childNode из currentNode.ДочернииеУзлы делать
             RewriteReferencesForNodesThatAreChildNodes
```

```
ToThoseThatContainCapturedVariable
(childNode, var.name,
 var.SyntaxTreeNodeWithVarDeclaration)
```

Рис. 10. Алгоритм обработки захваченных переменных (окончание)

Здесь `_capturedVarsClassDefs` – это коллекция всех сгенерированных синтаксических узлов классов для пространств имен.

Все шаги алгоритма, кроме шага 4, не нуждаются в дополнительных комментариях. Смысл шага 4 будет объяснен позже.

Прежде чем переходить к описанию алгоритма, реализуемого в методе `VisitCapturedVar`, который является ключевым на данном этапе анализа захваченных переменных, приведем некоторые вспомогательные факты.

При обходе древовидной структуры, о которой говорилось ранее, для каждого узла, отвечающего конкретному пространству имен, ставится в соответствие экземпляр класса `ScopeClassDefinition`. Это значит, что каждому анализируемому пространству имен, будь то блок операторов, подпрограмма или цикл, генерируется синтаксический узел, представляющий класс. Каждый экземпляр класса `ScopeClassDefinition` хранит следующую информацию:

1. поле `ClassDeclaration` типа `type_declaration`. Это генерируемый синтаксический узел-класс;
2. поле `CorrespondingSyntaxTreeNode` типа `syntax_tree_node`. Соответствует синтаксическому узлу, определяющему пространство имен, с которым данный экземпляр класса `ScopeClassDefinition` сопоставляется;
3. поле `GeneratedSubstitutingFieldName` – имя переменной, которая впоследствии будет использоваться для генерации синтаксического узла описания переменной. Таким образом, будет

генерироваться узел описания переменной типа `ClassDeclaration` с именем `GeneratedSubstitutingFieldName`;

4. поле `GeneratedUpperClassFieldName` – имя переменной-ссылки на объект генерируемого класса, связанного с пространством имен, являющимся непосредственным родителем текущего пространства имен. Это поле задействовано в том случае, когда в одном и том же лямбда-выражении захватывается несколько переменных, каждая из которых описана в своем пространстве имен. Тогда, чтобы обеспечить целостность ссылок, которая заключается в том, что для одной и той же захваченной переменной не создается несколько полей, принадлежащих разным сгенерированным классам, и что на разных уровнях-блоках замена обращений к этим захваченным переменным происходит корректным образом, с родительских пространств имен по цепочке передаются ссылки на объекты классов сгенерированных классов на дочерние;
5. поле `GeneratedVarStatementForScope` типа `var_statement`. Представляет собой узел описания переменной типа `ClassDeclaration` с именем `GeneratedSubstitutingFieldName`, о которой говорится в пункте 3;
6. поле `AssignNodeForUpperClassFieldInitialization` типа `assign`. Используется для передачи ссылок, о которых говорится в пункте 4.

Перейдем теперь к описанию алгоритма, реализуемого в методе `VisitCapturedVar`. Формальное описание алгоритма приведено на рис.11. Некоторые аспекты алгоритма, требующие дополнительных объяснений, будут прокомментированы ниже.

```
void VisitCapturedVar(CapturedVariablesTreeNode scope,
        CapturedVariablesTreeNode.CapturedSymbolInfo symbolInfo)
   varName := ПолучитьИмяПеременной(symbolInfo)
   Цикл от k := 0 до symbolInfo
                .КоличествоЗахватывающихПеременнуюЛямбд делать
     referencingLambda :=
     symbolInfo.ЗахватывающиеПеременнуюЛямбды[k]
     Если scope != referencingLambda.РодительскийУзел то
       upperScopeStack := ВычислитьПутьОтУзлаЛямбдыДоУзла
                              (referencingLambda, scope)
       upperScopeWhereVarsAreCaptured := scope
       upperScopeWhereVarsAreCapturedClass :=
         _capturedVarsClassDefs[upperScopeWhereVarsAreCaptured]
         .ClassDeclaration
       substKey := СоздатьКлючУзлаЗамены
                                  (varName, symbolInfo, scope)
       Если не _substitutions.СодержитКлюч(substKey) то
         _substitutions.ВычислитьИДобавить
                        УзелЗаменыДляКлюча(substKey)
       Пока не upperScopeStack.Пуст делать
         foundScopeWhereVarsWereCaptured := ложь
         Пока не upperScopeStack.Пуст и не
                        foundScopeWhereVarsWereCaptured делать
           Если upperScopeStack.Peek()
                .СодержитЗахваченныеПеременные делать
             foundScopeWhereVarsWereCaptured := истина
           Иначе
             curScope := upperScopeStack.Pop()
             substKey := СоздатьКлючУзлаЗамены
                               (varName, symbolInfo, curScope)
             Если upperScopeWhereVarsAreCaptured != scope то
               dotnode :=
               ВычислитьУзелЗаменыСУчетом
               ПередачиССылокСВерхнегоУровня
               НаНижние
               (upperScopeWhereVarsAreCaptured, scope, varName)
             Иначе
               dotnode :=
               СоздатьУзелЗамены
               (upperScopeWhereVarsAreCaptured, varName)
```

Рис. 11. Алгоритм `VisitCapturedVar`

```
             Если не _substitutions.СодержитКлюч(substKey) то
               _substitutions.
               ДобавитьУзелЗаменыДляКлюча(substKey, dotnode)
       Если foundScopeWhereVarsWereCaptured то
         nextNodeWhereVarsAreCaptured := upperScopeStack.Pop()
         Если не _capturedVarsClassDefs
                 .Содержит
```

```
            КлассДля(nextNodeWhereVarsAreCaptured) то
      nextNodeWhereVarsAreCapturedClass :=
        СоздатьКлассДля(nextNodeWhereVarsAreCaptured)
      _capturedVarsClassDefs
        .Добавить(nextNodeWhereVarsAreCapturedClass)
    Если _capturedVarsClassDefs
    [nextNodeWhereVarsAreCaptured].
    AssignNodeForUpperClassFieldInitialization==null то
      field :=
        СоздатьПолеДляХраненияСсылки
        НаОбъектКлассаВерхнегоПространстваИмен
        (upperScopeWhereVarsAreCapturedClass,
         nextNodeWhereVarsAreCaptured)
      nextNodeWhereVarsAreCapturedClass.
        ДобавитьПоле(field)
      _capturedVarsClassDefs
        [nextNodeWhereVarsAreCaptured].
        AssignNodeForUpperClassFieldInitialization =
          СоздатьУзелПрисваиванияДляПоляХраненияСсылки
          НаОбъектКлассаВерхнегоПространстваИмен
          (upperScopeWhereVarsAreCaptured,
           nextNodeWhereVarsAreCaptured)
    substKey := СоздатьКлючУзлаЗамены
      (varName, symbolInfo, nextNodeWhereVarsAreCaptured)
    dot :=
        ВычислитьУзелЗаменыСУчетом
        ПередачиССылокСВерхнегоУровня
        НаНижние
        (nextNodeWhereVarsAreCaptured, scope, varName)
    Если не _substitutions.СодержитКлюч(substKey) то
        _substitutions.
        ДобавитьУзелЗаменыДляКлюча(substKey, dot)
    upperScopeWhereVarsAreCaptured :=
      nextNodeWhereVarsAreCaptured
    upperScopeWhereVarsAreCapturedClass :=
      _capturedVarsClassDefs
      [upperScopeWhereVarsAreCaptured]
dotnode1 :=
  СформироватьУзелЗаменыДляПространстваИменЛямбды
  (upperScopeWhereVarsAreCaptured, varName)
```

Рис. 11. Алгоритм `VisitCapturedVar` (продолжение)

```
AddReferencesToIdentInLambda
 (referencingLambda, varName, symbolInfo,dotnode1);
referencingLambda
.ScopeIndexOfClassWhereLambdaWillBeAddedAsMethod :=
  ВычислитьИндексКлассаВКоторыйЛямбдаБудетДобавлена
  КакМетод(upperScopeWhereVarsAreCaptured)
```

```
Иначе
  Если не _capturedVarsClassDefs.СодержитКлассДля
                                  (scope) то
    cl := СоздатьКлассДля(scope)
    _capturedVarsClassDefs.Добавить(cl)
  substKey := СоздатьКлючУзлаЗамены
      (varName, symbolInfo, scope)
  Если не _substitutions.СодержитКлюч(substKey) то
    _substitutions.ВычислитьИДобавить
                  УзелЗаменыДляКлюча(substKey)
  referencingLambda
    .ScopeIndexOfClassWhereLambdaWillBeAddedAsMethod :=
      ВычислитьИндексКлассаВКоторыйЛямбдаБудетДобавлена
      КакМетод(scope)
Если не
_lambdasToBeAddedAsMethods.Содержит(referencingLambda) то
  _lambdasToBeAddedAsMethods.Добавить(referencingLambda)
```

Рис. 11. Алгоритм `VisitCapturedVar` (окончание)

Рассмотрим некоторые моменты алгоритма. Поле `_capturedVarsClassDefs` – это коллекция всех сгенерированных синтаксических узлов классов для пространств имен. Здесь по мере необходимости сохраняются все вновь сгенерированные классы для пространств имен. В коллекции `_substitutions` хранятся все синтаксические узлы замены для захваченных переменных, распределенные по ключам замены. Ключ замены, как говорилось ранее, представлен объектами класса `SubstitutionKey`. Поле `lambdasToBeAddedAsMethods` хранит все узлы лямбда-выражений, которые впоследствии будут добавлены как метод в один из сгенерированных классов. К какому именно классу будет добавлено то или иное лямбда-выражение, вычисляется отдельно, и обновленное значение записывается в свойство `ScopeIndexOfClassWhereLambdaWillBeAddedAsMethod`.

Метод `AddReferencesToIdentInLambda` вызывается для того, чтобы добавить замены на обращения к захваченным переменным в пространстве имен лямбда-выражений, так как вид замен внутри лямбда-выражений несколько отличается от вида замен в родительских по отношению к ним

пространствах имен. Этот метод рекурсивно вызывается для всех дочерних пространств имен лямбда-выражений.

Вернемся к объяснению шага 4 алгоритма на рис. 10. Метод `RewriteReferencesForNodesThatAreChildNodesToThoseThatContainCapturedVariable` вызывается для всех дочерних узлов пространств имен, в которых были захвачены переменные. Делается это потому, что может возникнуть ситуация, изображенная в примере 13.

```
begin
  var a := 1;
  begin
    a += 1;
    writeln(a);
  end;
  Seq(1,2,3).Select(x -> x + a).Print();
end.
```

Пример 13.

Здесь имеется блок, в котором описана переменная `a`, и дочерний по отношению к нему блок, в котором имеется ссылка на переменную `a`. При обработке захваченной переменной `a` в методе `VisitCapturedVar` будут обработаны лишь обращения к переменной `a`, относящиеся к этому родительскому блоку. Однако для получения корректной программы должны быть заменены *все* ссылки на переменную `a`, в том числе и в дочернем блоке. Этой задачей и занимается шаг 4 алгоритма из рис. 10 при вызове метода `RewriteReferencesForNodesThatAreChildNodesToThoseThatContainCapturedVariable`.

4.4.4 Преобразования узлов синтаксического дерева в результате замыканий

Здесь будет описан третий и последний этап анализа захваченных переменных в лямбда-выражениях, а именно обход сгенерированных на

предыдущем этапе синтаксических структур и замена всех ссылок на захваченные переменные.

4.4.4.1 NodesGenerator и его модификации в связи с введением в PascalABC.NET замыканий

При разработке компилятора PascalABC.NET используется набор утилит, каждая из которых призвана автоматизировать выполнение конкретной задачи. Одной из таких утилит является NodesGenerator.

NodesGenerator – приложение с графическим интерфейсом, которое позволяет создавать новые или изменять уже существующие определения узлов синтаксического дерева. Определения узлов представляют собой набор подузлов и методов того или иного узла. Утилита имеет внешний вид, изображенный на рис. 12.

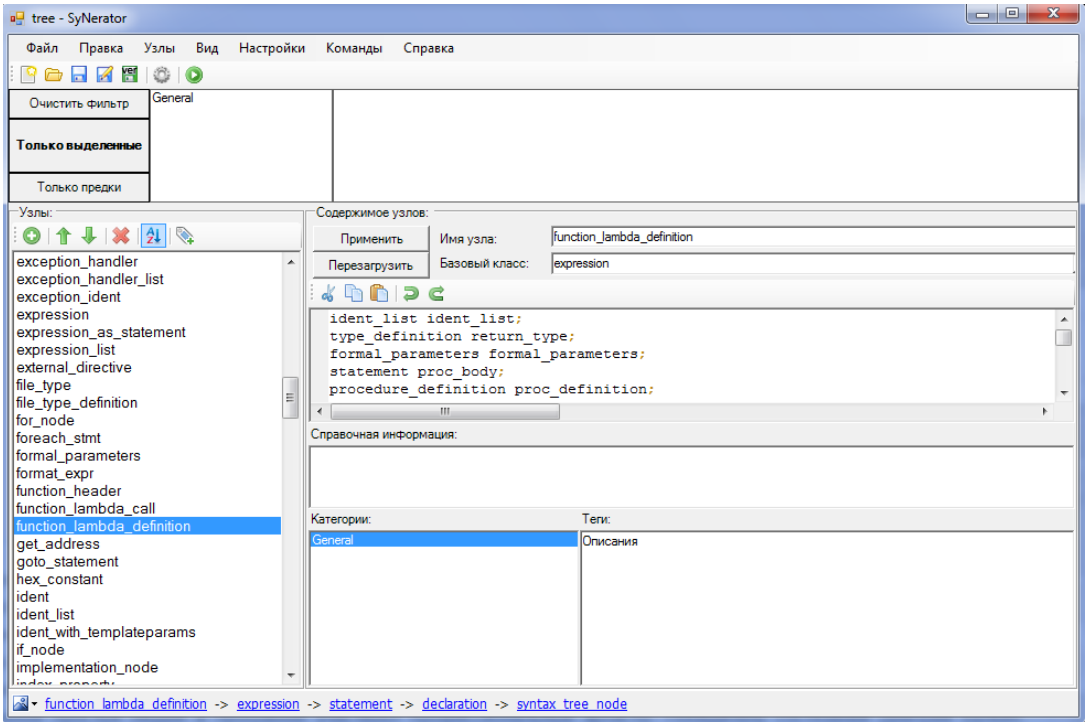

Рис.12. Утилита NodesGenerator

В связи с введением в PascalABC.NET поддержки захвата переменных в лямбда-выражениях NodesGenerator претерпел некоторые изменения. А именно был добавлен код, который для классов каждого синтаксического узла генерирует:

- `subnodes_count: int` – количество всех подузлов данного узла;
- индексатор, позволяющий обратиться по индексу, представленному целым числом, к любому подузлу данного узла.

Следует отметить, что если узел имеет подузлы, тип которых – список, то каждый элемент такого подузла считается также подузлом. Свойство `subnodes_count` и индексатор это учитывают.

В примере 14 приведен сгенерированный класс для узла `expression_list`.

```
public class expression_list : expression
{
     protected List<expression> _expressions =
                                        new List<expression>();
     public override Int32 subnodes_count
     {
          get
          {
               return 1 +
               (expressions == null ? 0 : expressions.Count);
          }
     }
     public override object this[Int32 ind]
     {
          get
          {
               if(subnodes_count == 0 || ind < 0
                    || ind > subnodes_count-1)
                    throw new IndexOutOfRangeException();
               switch(ind)
               {
                    case 0:
                         return expressions;
               }
```

Пример 14.

```
               Int32 index_counter=ind - 1;
               if(expressions != null)
               {
                    if(index_counter < expressions.Count)
                    {
                         return expressions[index_counter];
                    }
               }
```

```
                return null;
            }
            set
            {
                if(subnodes_count == 0 || ind < 0
                    || ind > subnodes_count-1)
                    throw new IndexOutOfRangeException();
                switch(ind)
                {
                    case 0:
                        expressions = (List<expression>)value;
                        break;
                }
                Int32 index_counter=ind - 1;
                if(expressions != null)
                {
                    if(index_counter < expressions.Count)
                    {
                        expressions[index_counter]=
                            (expression)value;
                        return;
                    }
                }
            }
        }
}
```

Пример 14 (окончание).

Отметим, что свойство `subnodes_count` и индексатор объявлены виртуальными в базовом для всех синтаксических узлов классе и переопределяются в каждом синтаксическом узле.

Свойство `subnodes_count` и индексатор генерируются для достижения следующей цели. Замена обращений к переменным, захваченным в лямбда-выражениях, происходит на синтаксическом уровне, то есть заменяются узлы синтаксического дерева. На самом деле узлы типа `ident` заменяются на узлы типа `dot_node`. При замене обращений в данном узле необходим общий доступ к подузлам этого узла. То есть необходим способ обхода всех подузлов как коллекции и в случае необходимости подставлять вместо узла типа `ident` узел типа `dot_node`. Свойство `subnodes_count` позволяет получить количество элементов в этой коллекции, а индексатор – запустить цикл, в

котором для каждого подузла проверяется, имеет ли он тип `ident`, а также нужно ли производить замену этого подузла.

Данный подход определяет общую логику обхода подузлов всех узлов, что является очень удобным, так как не приходится описывать способ обхода для каждого типа узла в отдельности.

4.4.4.2 Семантический анализ сгенерированных структур. Замена ссылок на захваченные переменные

Теперь, когда собрана вся необходимая информация о захваченных переменных и контексте их описания и сгенерированы соответствующие синтаксические структуры, начинается выполнение последнего этапа обработки захваченных переменных в лямбда-выражениях – непосредственный обход этих синтаксических структур и замена ссылок на захваченные переменные.

Для этого автором работы был разработан класс `CapturedVariablesSubstitutor`, который является наследником класса `WalkingVisitor`, рассмотренного ранее. Ниже опишем основные действия, которые выполняет этот класс в процессе анализа.

Обход всех узлов и замена ссылок на захваченные переменные.

Как уже говорилось, класс `WalkingVisitor` описывает событие `OnLeave`, которое активизируется в момент завершения обхода конкретного узла. Замена всех обращений к захваченным переменным происходит в этот момент. Стоит отметить, что этот момент выбран не случайно: заменять одни узлы на другие нужно тогда, когда все подузлы конкретного узла уже проанализированы и с уверенностью можно сказать, что замена прошла корректно также в подузлах этого узла. В этом случае будут обработаны *все* обращения к захваченным переменным, так что ни одно из них не будет пропущено.

Алгоритм замены ссылок на захваченные переменные основан на следующих предположениях. В процессе обхода той части синтаксического дерева, в которой производится замена, сохраняется узел текущего просматриваемого блока операторов типа `syntax_tree_node` (назовем его `currentBlockNode`), а также узел, в котором происходит замена (назовем его `nodeWhereSubstitute`). Далее из `nodeWhereSubstitute` выбираются все подузлы типа `ident` – потенциальные обращения к захваченным переменным. Этому способствует свойство `subnodes_count` и индексатор узла, которые были описаны в предыдущем разделе. В `CapturedVariablesSubstitutor` в качестве входных данных приходит список `_identsReferences` всех ссылок на все идентификаторы (потенциально заменяемые узлы) в части синтаксического дерева, переданной на анализ, а также словарь `_substitutionsInfo`, ключом которого является объект типа `SubstitutionKey`, а значением – узел типа `dot_node`, представляющий собой замену, соответствующую конкретному `SubstitutionKey`. Подузлы типа `ident` узла `nodeWhereSubstitute` сопоставляются с узлами из списка `_identsReferences`, и для каждого такого идентификатора строится объект типа `SubstitutionKey`. При построении такого объекта участвуют также `currentBlockNode`. После того как этот объект-ключ построен, происходит поиск замены для данного идентификатора в `_substitutionsInfo` по этому ключу. Если для этого идентификатора существует заменяющий узел, то происходит замена узла типа `ident` на узел типа `closure_substituting_node`, который хранит в себе этот заменяющий узел типа `dot_node`. При финальном обходе части синтаксического дерева, переданной на анализ, этот узел обходится вместо первоначального `ident`.

Таким образом, после полного обхода этой части синтаксического дерева все ссылки на захваченные переменные будут корректно заменены.

Замена узлов лямбда-выражений.

На данный этап анализа захваченных переменных в качестве входных данных приходит список узлов лямбда-выражений `_lambdasToBeAddedAsMethods`, а также структура `_generatedScopeClassesInfo` с информацией о сгенерированных классах, в которые эти лямбда-выражения должны быть добавлены как методы. Эти данные передаются со второй стадии обработки замыканий.

На данном этапе происходит сопоставление лямбда-выражения и класса, в который оно добавляется как метод. Также необходимо учесть, что из программы обращение к этому методу будет иметь вид `имя_объекта_класса.имя_лямбда_выражения`, поэтому на данном этапе также генерируется соответствующая замена для лямбда-выражений, подобная тем, что рассмотрены в пункте «*Обход всех узлов и замена ссылок на захваченные переменные*», и записывается в поле `substituting_node` узла `function_lambda_definition`, которое было добавлено специально для этих целей.

Семантический анализ сгенерированных классов.

Здесь происходит обход сгенерированных синтаксических узлов-классов из `_generatedScopeClassesInfo` с целью преобразования их в узлы семантического дерева.

При обходе интенсивно задействован класс `compilation_context`, хранящий информацию о текущем контексте компиляции. Под контекстом компиляции понимаются все данные о стеке функций, всех разобранных типах и т.д. Для нас важную роль играет такая информация, хранящаяся в `compilation_context`, как стек пространств имен. Дело в том, что на момент, когда необходимо обойти сгенерированные классы и преобразовать их в узлы семантического дерева, контекст компиляции находится в состоянии, когда текущим пространством имен является тот блок операторов, для которого

производится анализ захваченных переменных. Однако информация о типах, в том числе классах, должна быть добавлена на глобальный уровень. Поэтому нужно иметь возможность переходить в `compilation_context` на глобальный уровень, сохранив при этом контекст компиляции в том виде, в котором он находится на данный момент, произвести семантический анализ сгенерированных классов, и вернуть контекст в первоначальное состояние. Это достигается с помощью методов `SaveContextAndUpToGlobalLevel` и `RestoreCurrentContext` класса `compilation_context`.

Замена узлов описания захваченных переменных.

После того как была произведена замена захваченных переменных, в синтаксическом дереве программы появились ссылки на объекты сгенерированных классов. Однако эти переменные в модифицированном дереве еще не были объявлены.

Этой задачей занимается данный этап анализа. В процессе его выполнения происходит поиск всех узлов, в которых были описаны захваченные переменные, удаляется информация об их описании, и наконец, генерируются узлы типа `var_def_statement`, которые вставляются в необходимое место в синтаксическом дереве. Если для текущего пространства имен был предусмотрен узел `AssignNodeForUpperClassFieldInitialization` (см. 4.4.3), то он также помещается на нужный уровень в синтаксическое дерево.

4.4.5 Специальные случаи замыканий

В данном разделе будут приведены примеры различных случаев захвата переменных. Это позволит, с одной стороны, подытожить рассмотрение реализации захвата переменных, а с другой – на примерах показать результат выполнения стадий анализа, описанных на протяжении нескольких предыдущих разделов.

Так как захват локальных переменных был уже в той или иной мере рассмотрен в разделе 4.4.3 (см. пример 12) и представляет поэтому меньший интерес, в этом разделе будут освещены следующие виды захвата:

- Замыкание параметров подпрограмм;
- Замыкание переменных циклов;
- Замыкание полей классов;
- Замыкания и шаблонные подпрограммы;
- Замыкание нескольких видов переменных.

4.4.5.1 Захват параметров подпрограмм

Рассмотрим ситуацию, когда лямбда-выражение используется внутри подпрограммы и происходит захват параметра этой подпрограммы внутри этого лямбда-выражения. В коде примера 15 приведена простая программа, иллюстрирующая данную ситуацию.

Замечание: здесь и далее в части (а) примеров будет приводиться исходный код программы, а в части (б) – код, генерируемый компилятором после обработки замыканий и представляющий декомпилированный IL-код на язык C#.

```
function IsPrime(x: integer): boolean;
begin
  Result := Range(2,Round(sqrt(x))).All(i->x mod i <> 0)
end;
```

Пример 15(а).

```
begin
  writeln(IsPrime(4));
  writeln(IsPrime(13));
  writeln(IsPrime(17));
end.
```

Пример 15(а, окончание).

```
public class <>local_variables_class_0
{
     public int32 x;
     public void <>local_variables_class_0(int32 _x)
     {
          this.x = _x;
     }
     public bool <>lambda1(int32 i)
     {
          return (this.x % i);
     }
}
public class Program
{
     public static bool IsPrime(int32 x)
     {
          <>local_variables_class_0 <>local_variables_class_01 =
                                    <>local_variables_class_0(x);
          return Range(2,Round(sqrt(x))).
                 All(new Func<int, bool>
                    (<>local_variables_class_01.<>lambda1));
     }
     public static void Main()
     {
          writeln(IsPrime(4));
          writeln(IsPrime(13));
          writeln(IsPrime(17));
     }
}
```

Пример 15(б).

В данном примере показано, что при захвате параметра подпрограммы для пространства имен этой подпрограммы генерируется класс, и в этот класс помещается поле, соответствующее этой захваченной переменной. Стоит отметить, что для инициализации этого поля начальным значением генерируется конструктор, вызываемый при создании объекта этого класса.

4.4.5.2 Захват переменной цикла

В данном разделе рассматриваются особенности захвата итерационных переменных: переменных цикла `for` и `foreach`.

Захват переменной цикла `for`.

При захвате итерационной переменной цикла `for` в PascalABC.NET она трактуется так, как будто бы объявлена за пределами цикла. Это значит, что на каждой итерации захватывается та же самая переменная. Приведенная в примере 16 программа выведет 333 вместо 123.

Лямбда-выражения, создаваемые на отдельных итерациях цикла, захватывают ту же самую переменную `i`. Это действительно имеет смысл, если считать, что `i` является переменной, значение которой сохраняется между итерациями цикла. Как следствие, когда делегаты вызываются впоследствии, каждый делегат видит значение `i` на момент вызова, т.е. 3.

```
begin
  var actions := new List<System.Action>();
  for var i := 1 to 3 do
    actions.Add(procedure() -> write(i));
  foreach var action in actions do
    action();
end.
```

Пример 16(а).

```
public class <>local_variables_class_0
{
    public int32 i;
    public void <>lambda1()
     {
          write(this.i);
     }
}
```

Пример 16(б).

```
public class Program
{
    public static void Main()
     {
          List<System.Action> actions =
                         new List<System.Action>();
          <>local_variables_class_0 <>local_variables_class_01 =
                         new <>local_variables_class_0();
          for (int i = 1; i <= 3; i++)
          {
```

```
                <>local_variables_class_01.i = i;
                actions.Add(
                new Action(<>local_variables_class_01.<>lambda1));
            }
            foreach var action in actions do
                action();
        }
}
```

Пример 16(б, окончание).

Захват переменной цикла `foreach`.

Изменим немного программу из примера 16, заменив первый цикл `for` на эквивалентный ему цикл `foreach` (см. пример 17). На этот раз вывод, который мы получим в результате выполнения программы, будет не 333, а 123.

Объясняется это тем, что на каждой итерации создается собственный экземпляр класса замыкания, который содержит поле `i`. Такой подход обосновывается тем, что в отличие от переменной цикла `for`, переменная цикла `foreach` на каждой итерации может рассматриваться как отдельный независимый элемент перебираемой коллекции, и поэтому на каждой итерации логичнее было бы захватывать именно конкретное значение этой переменной.

```
begin
  var actions := new List<System.Action>();
  foreach var i in Seq(1,2,3) do
    actions.Add(procedure() -> write(i));
  foreach var action in actions do
    action();
end.
```

Пример 17(а).

```
public class <>local_variables_class_0
```

```
{
    public int32 i;
    public void <>lambda1()
     {
          write(this.i);
     }
}


public class Program
{
    public static void Main()
     {
          List<System.Action> actions =
                         new List<System.Action>();
          foreach (int i in Seq(1,2,3))
          {
               <>local_variables_class_0
               <>local_variables_class_01 =
                    new <>local_variables_class_0();
               <>local_variables_class_01.i = i;
               actions.Add(
               new Action(<>local_variables_class_01.<>lambda1));
          }
          foreach var action in actions do
               action();
     }
}
```

Пример 17(б).

4.4.5.3 Захват полей класса

При захвате лямбда-выражением только лишь поля класса это лямбда-выражение просто добавляется как метод этого класса.

Наиболее интересен случай, когда происходит захват приватного поля класса и локальной переменной. В примере 18 иллюстрируется данная ситуация.

Как видно, для пространства имен, соответствующего блоку, в котором описана переменная `t`, был сгенерирован класс `<>local_variables_class_0`. Так как вместе с переменной `t` захватывается поле `a` класса `cl`, то в `<>local_variables_class_0` должна иметься ссылка на класс пространства

имен, соответствующий классу `cl`. Эта ссылка представлена полем `<>local_variables_class_UPPER_0`.

После преобразований синтаксического дерева в сгенерированном методе `<>lambda1` происходит обращение к полю `a` класса `cl`. Так как поле `a` приватное, то это по правилам языка должно приводить к ошибке компиляции, так как к приватным полям одного класса запрещено обращаться из другого класса. Чтобы обойти данное ограничение, было принято решение генерировать дополнительные свойства для каждого такого приватного поля, которое позволяет обращаться к этим полям через них (в данном случае, для поля `a` сгенерировано свойство `<>nonPublica0`). Имена для таких свойств генерируются тем же способом, что и для имен лямбда-выражений, генерируемых классов и т.д. Сделано это для гарантии отсутствия конфликтов имен.

```
type cl = class
  private
    a: integer;
  public
    constructor Create(_a: integer);
    begin
      a := _a;
    end;
```

Пример 18(а).

```
    function f(x: integer): System.Action;
    begin
      var t := 1;
      a += x;
      writeln(a);
      result := procedure() ->
                begin
                  a += t;
                  writeln(a);
                end;
    end;
end;
```

```
begin
  var c := new cl(5);
  var actions := new List<System.Action>();
  actions.Add(c.f(1));
  actions.Add(c.f(3));
  actions[0]();
  actions[1]();
end.
```

Пример 18(а, окончание).

```
public class <>local_variables_class_0
{
    public cl <>local_variables_class_UPPER_0;
    public int32 t;

    public void <>lambda1()
     {
          this.<>local_variables_class_UPPER_0.<>nonPublica0 =
          this.<>local_variables_class_UPPER_0.<>nonPublica0 +
          this.t;
          writeln(this.<>local_variables_class_UPPER_0.
                  <>nonPublica0);
     }
}

public class cl
{
     private int32 a;
     public int32 <>nonPublica0
     {
          get { return this.a; }
          set { this.a = value; }
     }
```

Пример 18(б).

```
     public void cl(int32 _a)
     {
          this.a = _a;
     }

     public Action f(int32 x)
     {
          <>local_variables_class_0 <>local_variables_class_01 =
                    new <>local_variables_class_0();

          <>local_variables_class_01.
```

```
            <>local_variables_class_UPPER_0 = this;

            <>local_variables_class_01.t = 1;

            <>local_variables_class_01.
            <>local_variables_class_UPPER_0.a =
                 <>local_variables_class_01.
                 <>local_variables_class_UPPER_0.a + x;
            writeln(<>local_variables_class_01.
                 <>local_variables_class_UPPER_0.a);

            return
                 new Action(<>local_variables_class_01.<>lambda1);
      }
}

public class Program
{
      cl c = new cl(5);
      List<System.Action> actions = new List<System.Action>();
      actions.Add(c.f(1));
      actions.Add(c.f(3));
      actions[0]();
      actions[1]();
}
```

Пример 18(б, окончание).

4.4.5.4 Захват переменных и шаблонные подпрограммы

В случае, когда лямбда-выражения используются внутри шаблонных подпрограмм или классов, то методы, генерируемые для этих лямбда-выражений, также будут шаблонными. При этом если в заголовках этих подпрограмм и классов в секции `where` описаны ограничения на параметры типов, то они должны быть также перенесены в методы, генерируемые для лямбда-выражений (в случае захвата переменных – в заголовки генерируемых классов).

Все вышесказанное описывает пример 19.

```
function f<T>(e: IEnumerable<T>; b: T): IEnumerable<T>;
                                        where T: class;
begin
   result := e.OrderBy(x->x.GetHashCode() xor b.GetHashCode());
```

```
end;

begin
  writeln(f(Seq('aaa', 'daa', 'abc', 'bca'), 'abcde'));
end.
```

Пример 19(а).

```
public class <>local_variables_class_0<T> where T : class
{
     public T b;
     public void <>local_variables_class_0(T _b)
     {
          this.b = _b;
     }
     public int32 <>lambda1(T x)
     {
          return x.GetHashCode() ^ this.b.GetHashCode();
     }
}

public class Program
{
     public static IEnumerable<T> f(IEnumerable<T> e, T b)
                                          where T : class
     {
          <>local_variables_class_0<T>
               <>local_variables_class_0 =
               new <>local_variables_class_0<T>(b);
          return e.OrderBy(
                    new Func<T, int>
                         (<>local_variables_class_0.<>lambda1));
     }
```

Пример 19(б).

```
     public static void Main()
     {
          writeln(f(Seq("aaa", "daa", "abc", "bca"), "abcde"));
     }
}
```

Пример 19(б, окончание).

4.4.5.5 Захват нескольких видов переменных

Этот пункт завершает главу 4. Ниже будет приведен пример программы (см. пример 20), в которой происходит захват нескольких видов переменных. Пример позволяет в полной мере проиллюстрировать результат работы алгоритмов, описанных в главе. Стоит обратить внимание на то, как происходит передача ссылок на объекты с более высоких уровней на более низкие.

```
procedure pr(a: integer);
begin
  var b: integer := 2;
  begin
    var c: integer := 3;
    begin
      var d: integer := 4;
      foreach var t in Seq(1,2,3,4) do
      begin
        writeln(Range(1, t).Select(x -> x + a + b + d + t));
        writeln(t);
      end;
      d += 1;
      a *= 2;
      writeln(d);
      writeln(a);
    end;
  end;
  begin
    writeln(Range(1, 10).Select(x -> x + a + b));
    writeln(a);
  end;
end;

begin
  pr(5);
end.
```

Пример 20(a).

```
public class <>local_variables_class_0
{
     public int32 a;
     public void <>local_variables_class_0(int32 _a)
     {
          this.a = _a;
     }
}

public class <>local_variables_class_1
{
```

```
     public <>local_variables_class_0
          <>local_variables_class_UPPER_0;
     public int32 b;
     public int32 <>lambda2(int32 x)
     {
          return x + this.<>local_variables_class_UPPER_0.a +
                 this.b;
     }
}

public class <>local_variables_class_2
{
    public <>local_variables_class_1
          <>local_variables_class_UPPER_1;
    public int32 d;
}

public class <>local_variables_class_3
{
     public <>local_variables_class_2
          <>local_variables_class_UPPER_2;
     internal int32 t;
     public int32 <>lambda1(int32 x)
     {
          return x +
                 this.<>local_variables_class_UPPER_2.
                    <>local_variables_class_UPPER_1.
                    <>local_variables_class_UPPER_0.a +
                 this.<>local_variables_class_UPPER_2.
                     <>local_variables_class_UPPER_1.b +
                 this.<>local_variables_class_UPPER_2.d +
                 this.t;
     }
}
```

Пример 20(б).

```
public class Program
{
     public static void pr(int32 a)
     {
          <>local_variables_class_0 <>local_variables_class_01 =
               new <>local_variables_class_0(a);
          <>local_variables_class_1 <>local_variables_class_11 =
               new <>local_variables_class_1();
          <>local_variables_class_11.
          <>local_variables_class_UPPER_0 =
```

```
     <>local_variables_class_01;
<>local_variables_class_11.b = 2
{
     int c = 3;
     {
          <>local_variables_class_2
               <>local_variables_class_21 =
               new <>local_variables_class_2();
          <>local_variables_class_21.
          <>local_variables_class_UPPER_1 =
               <>local_variables_class_11;
          <>local_variables_class_21.d = 4;

          foreach (int t in Seq(1,2,3,4))
          {
               <>local_variables_class_3
                    <>local_variables_class_31 =
                    new <>local_variables_class_3();
               <>local_variables_class_31.
               <>local_variables_class_UPPER_2 =
                    <>local_variables_class_21;
               <>local_variables_class_31.t = t;

               writeln(
                    Range(1,
                    <>local_variables_class_31.t).
                    Select(
                    new Func<int, int>(
                    <>local_variables_class_31.
                    <>lambda1)));
               writeln(<>local_variables_class_31.t);
          }

          <>local_variables_class_21.d =
               <>local_variables_class_21.d + 1;
```

Пример 20(б, продолжение).

```
          <>local_variables_class_21.
          <>local_variables_class_UPPER_1.
          <>local_variables_class_UPPER_0.a =
               <>local_variables_class_21.
               <>local_variables_class_UPPER_1.
               <>local_variables_class_UPPER_0.a * 2;

          writeln(<>local_variables_class_21.d);
          writeln(<>local_variables_class_21.
               <>local_variables_class_UPPER_1.
               <>local_variables_class_UPPER_0.a);
```

```
            }
        }
        {
            writeln(Range(1, 10).Select(
            new Func<int, int>(
            <>local_variables_class_11.<>lambda2));
            writeln(<>local_variables_class_11.
                <>local_variables_class_UPPER_0.a);
        }
    }
    public static void Main()
    {
        pr(5);
    }
}
```

Пример 20(б, окончание).

Заключение

В диссертационной работе были описаны средства введения замыканий в язык PascalABC.

В процессе работы над реализацией замыканий были решены следующие задачи:

1. реализованы замыкания в PascalABC.NET с поддержкой следующих видов захвата переменных:
   - захват локальных переменных;
   - захват параметров процедур и функций;
   - захват переменных циклов `for` и `foreach`;
   - захват полей класса, включая поля, не являющиеся публичными;
2. обеспечена поддержка одновременного использования нескольких видов захвата;
3. реализована поддержка лямбда-выражений внутри шаблонных подпрограмм;
4. модифицирован алгоритм вывода параметров типов при инстанцировании шаблонных подпрограмм при реализации лямбда-выражений.

Популярность лямбда-выражений в современных языках программирования обусловлена удобным синтаксисом, позволяющим описывать подпрограмму там, где ее использование ограничивается одним вызовом, т.е. при ее передаче и возвращении в качестве возвращаемого значения подпрограмм. Возможность не указывать явно типы параметров и возвращаемого значения, а также возможность захвата переменных из внешнего контекста значительно сокращают трудозатраты программиста.

Поддержка замыканий позволила расширить язык PascalABC.NET, открыв доступ к таким мощным средствам платформы .NET, как библиотека Linq.

Литература

1. Сайт проекта PascalABC.NET – http://pascalabc.net
2. Ткачук А.В. Язык, компилятор и система программирования PascalABC.NET. Дипломная работа, Южный Федеральный Университет, 2007.

3. Иванов С.О. Языковые средства и генерация кода шаблонов классов для PascalABC.NET. Магистерская диссертация, Южный Федеральный Университет, 2008.
4. Сайт справочных материалов корпорации Microsoft - https://msdn.microsoft.com
5. Ж. Довек, Ж.-Ж. Леви. Введение в теорию языков программирования. М.: ДМК Пресс, 2013.
6. Д. Рихтер. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. СПб.: Русская Редакция; СПб.: Питер, 2015.
7. Д. Албахари, Б. Албахари. C# 5.0. Справочник. Полное описание языка. М.: Вильямс, 2014.
8. Водолазов Н.Н. Конвертор в семантическое дерево для компилятора PascalABC.NET. Магистерская диссертация, Южный Федеральный Университет, 2007.
9. Б. Страуструп. Дизайн и эволюция языка C++. М.: ДМК Пресс, 2014.
10. Bjarne Stroustrup. C++11 - the new ISO C++ standard. http:// http://www.stroustrup.com/C++11FAQ.html