# РЕАЛИЗАЦИЯ ОПЕРАТОРА YIELD ПО СИНТАКСИЧЕСКОМУ ДЕРЕВУ

Баташов О.А.
4 курс 9 группа
Научный руководитель:
Михалкович С.С.

# Постановка задачи

- Реализовать оператор **yield** в PascalABC.NET
- *Какие возможности это даёт?*
- Возможность описывать **методы-итераторы**
- Метод, в котором используется ключевое слово ***yield*** для перебора по коллекции или массиву

```
function Gen: sequence of integer;
begin
  yield 1;
  yield 5;
end;

begin
  var q := Gen();
  foreach var x in q do
    Print(x);
end.
```

# Как это реализуется

- Синтаксический сахар
- Тело метода преобразуется в конечный автомат (в несколько этапов), сохраняющий состояние метода-итератора между вызовами
- Автомат реализуется в вспомогательном классе, реализующем интерфейс IEnumerable

```
public static IEnumerable<int> Gen()
{
    return new clyield#1();
}
```

```
public bool MoveNext()
{
    bool result;
    switch (this.<>1__state)
    {
    case 0:
        this.<>2__current = 1;
        this.<>1__state = 1;
        result = true;
        break;
    case 1:
        this.<>2__current = 5;
        this.<>1__state = 2;
        result = true;
        break;
    }
    return result;
}
```

# Основные проблемы

- Сохранение состояния между вызовами метода-итератора
- Проблема циклических (**нелинейных**) операторов (*for, while, repeat..until*)
- Тело метода должно иметь **линейную** структуру
- Lowering (**развертка**: преобразование нелинейного кода в линейный)

# Lowering: пример

- Циклическая конструкция (псевдокод)

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

- Превращается в линейную последовательность операторов - можно сделать goto к телу цикла

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

# Lowering: проблемы

```
for var i := 1 to 777 do
```

- Разворачивается в
  - **var <>LV_i;**
  - lowered-code
- А если в коде присутствует несколько таких for?

```
for var i := 1 to 777 do
for var i := 1 to 777 do
```

- Проблема повторяющихся имен
- Решение: использовать уникальные имена, такие как var <>LV_i**#1**

# Lowering: проблема не приходит одна

```
var i := 666;
for var i := 1 to 777 do // Повторное объявление!
```

- Преобразование ***i*** в ***<>LV_i#1*** нивелирует ошибку и такой код компилируется!
- Решение: отслеживать такие ошибки (спец.визитор) ДО выполнения lowering

# Lowering сделан. Что дальше?

- Необходимость сохранения состояния между вызовами метода-итератора приводит к необходимости захвата имён
- Разный способ захвата в зависимости от класса имени
- Классификация имён:
  - Локальные переменные
  - Формальные параметры метода
  - Имена класса, в котором описан метод (если он описан в классе)
  - Глобальные имена

# Захват локальных переменных и формальных параметров

- Локальные переменные и формальные параметры метода становятся полями вспомогательного класса-автомата
- В теле метода обращения к этим именам заменяются на обращения к полям вспомогательного класса

```
function Gen(n: integer): sequence of integer;
var j,k: real;
begin
  var i := 1;
  j := n;
  while i<j do
  begin
    yield i*i;
    i += 1;
  end;
end;

begin
  foreach var x in Gen(10) do
    Print(x);
end.
```

```
case 0:
    this.<j>MethodLocalVariable__1 = (double)this.<>MethodFormalParam__n;
    break;
case 1:
    this.<i>MethodLocalVariable__3++;
    break;
```

# Проблема: снова повторяющиеся имена

```
begin
  var i := 777.7;
end;
begin
  var i := 666.6; // На самом деле,
end;
```

```
begin
  var i_1 := 777.7;
end;
begin
  var i_2 := 666.6;
end;
```

- Мини-пространства имен
- Решение:
  - Выполнить переименование перед lowering
  - Выполнить удаление лишних **begin..end** после lowering

# Захват локальных переменных: проблема типов

- Для захвата локальной переменной как поля вспомогательного класса необходимо знать ее тип
- ***var x := 666.6;*** *// Тип x очевиден? На самом деле, не очень!*
- Синтаксический сахар внедряется до перевода синтаксического дерева в семантическое, определить тип переменной невозможно (по крайней мере, легко и быстро)
- Решение: “вязкая семантика”
- Определение типа откладывается до этапа семантики, используя вспомогательные узлы синтаксического дерева

# Захват имен класса, содержащего метод-итератор

- Если метод определен в классе, выполняем захват self этого класса и обращаемся к именам через него

```
type A = class
  testField: real := 777.7;

  function Gen: sequence of real;
  begin
      yield testField;
  end;
end;
```

```
this.<>2__current = this.<>4__self.testField;
this.<>1__state = 4;
result = true;
return result;
```

# Захват имен класса: проблема базового класса

- *Проблема на этапе синтаксиса*
- Невозможно определить принадлежность имени базовому классу, находящемуся в .NET сборке
- Решение аналогично - прибегнуть к “вязкой семантике” и вспомогательным синтаксическим узлам

# Алгоритм формирования конечного автомата для метода-итератора

1. Выполнить lowering тела метода
2. Найти все идентификаторы в теле метода и отклассифицировать их
3. Выполнить захват:
    a. Локальные переменные поднять как поля вспомогательного класса и удалить их описания из тела
    b. Формальные параметры поднять как поля вспомогательного класса
    c. Имена, принадлежащие классу, обернуть через захваченный self класса.
    d. Остальные имена оставить без изменений
4. На месте тела метода создать вспомогательный класс, реализующий интерфейс IEnumerable и добавить в него поля из шага (2)
5. Сформировать конечный автомат в методе MoveNext вспомогательного класса
    a. Количество yield = количество состояний + начальное (0)
    b. Сформировать секцию case по состояниям + обработка начального (0) состояния
    c. Последовательно обходить операторы, помещая их в секцию case для обрабатываемого состояния
    d. Если встречен оператор yield, заменить его на последовательность операторов:
        i. current := <yielded-value>
        ii. state := <next-state>
        iii. return true.
    e. Если это не последний оператор, то сформировать новое состояние в case, иначе - закрыть case
    f. Если встречен оператор, помеченный меткой, то добавить в секцию после case метку и последовательность операторов до конца процедуры/до следующего yield. Если не в case встречен оператор, помеченный меткой, то его не обрабатываем, оставляем на том же месте.

# Пример

```
C:\Users\Oleg>"C:\Users\Oleg\Documents\Visual Studio 2015\Projects\C#\Compilers\
PascalABC.NET\Yield\tests\yieldDemo.exe"
5 38.3 5 777.7 77.3 83.9 0.00159265291648683 1 2 3 4 5
```

```
type BaseClass = class
  testBaseField: real := 77.3;

  function testBaseFunction(x: real): real;
  begin
    result := 83.9;
  end;
end;

type A = class(BaseClass)
  testField: real := 777.7;

  function Gen(testFormalParam: integer): sequence of real;
  var testLocalVariable := testFormalParam;
  begin
      var testLocalVariable_2 : real := testLocalVariable + 33.3;

      yield testLocalVariable;
      yield testLocalVariable_2;

      yield testFormalParam;

      yield testField;
      yield testBaseField;
      yield testBaseFunction(9);

      yield sin(3.14);

      for var x := 1 to testLocalVariable do
      begin
        yield x;
      end;
  end;
end;
```

# Сахар? Взрыв синтаксиса!

```
public class clyield#1Helper : IEnumerator, IEnumerable
{
    public int <$testLocalVariable__0>MethodLocalVariable__1;
    public double <$testLocalVariable_2__0>MethodLocalVariable__2;
    public int <$x1>MethodLocalVariable__3;
    public int <>MethodFormalParam__testFormalParam;
    public A <>4__self;
    public int <>1__state;
    public double <>2__current;
    public void $Init$()...
    public clyield#1Helper()...
    public void Reset()...
    public bool MoveNext()
    {
        bool result;
        switch (this.<>1__state)
        {
        case 0:
            this.<$testLocalVariable__0>MethodLocalVariable__1 = this.<>MethodFormalParam__testFormalParam;
            this.<$testLocalVariable_2__0>MethodLocalVariable__2 = (double)this.<$testLocalVariable__0>MethodLocalVariable__1 + 
            this.<>2__current = (double)this.<$testLocalVariable__0>MethodLocalVariable__1;
            this.<>1__state = 1;
            result = true;
            return result;
        case 1:
            this.<>2__current = this.<$testLocalVariable_2__0>MethodLocalVariable__2;
            this.<>1__state = 2;
            result = true;
            return result;
        case 2:
            this.<>2__current = (double)this.<>MethodFormalParam__testFormalParam;
            this.<>1__state = 3;
            result = true;
            return result;
        case 3:
            this.<>2__current = this.<>4__self.testField;
```

# Результаты

- Разработан и реализован алгоритм развертки (lowering) циклических конструкций
- Определена классификация имен в методе-итераторе и реализован алгоритм их классификации
- Реализован захват имен в теле метода-итератора
- Реализовано создание конечного автомата по методу-итератору

# Спасибо за внимание!

Код проекта

https://github.com/PascalABC-CompilerLaboratory/pascalabcnet/tree/master/Yield

Эта презентация

https://goo.gl/MbwNBV