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

Чумакова Евгения Геннадьевна

ПРЕОБРАЗОВАНИЯ СИНТАКСИЧЕСКИХ ДЕРЕВЬЕВ ПРИ РЕАЛИЗАЦИИ СИНТАКСИЧЕСКОГО САХАРА В PASCALABC.NET

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ
по направлению 01.04.02 – 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 – 2017

Оглавление

Введение 3

Постановка задачи 6

Глава 1. Синтаксический сахар 7

1.1 Основные понятия и примеры 7

1.2 Сахарные конструкции в разных языках программирования 8

Глава 2. Реализация «сахара» в PascalABC.NET 12

2.1 Общая структура компилятора PascalABC.NET 12

2.2 Визиторы PascalABC.NET 14

2.3 Принцип введения сахарных конструкций в PascalABC.NET 16

2.4 Проблема семантических проверок 19

Глава 3. Введение NULL-условного оператора — ?. 22

3.1 Изменение грамматики языка 22

3.2 DESUGARING 26

3.3 Семантические проверки 31

Глава 4. Введение оператора объединения со значением NULL — ?? 35

4.1 Изменение грамматики языка 35

4.2 DESUGARING 37

4.3 Проблема повторяющихся вычислений 37

Глава 5. Тестовые программы и их результаты 40

Заключение 46

Литература 47

Введение

В современном программировании все чаще встречается задача оптимизации процесса написания кода за счет использования синтаксических возможностей языка. И вполне закономерным является то, что с каждым годом спектр этих возможностей расширяется путем внедрения новых конструкций в структуру компилятора.

Компилятор – это программа, которая выполняет перевод текста программы, написанного на исходном языке программирования, в эквивалентный, но уже на целевом языке.

Описанный выше процесс – это и есть компиляция. Обычно она состоит из нескольких этапов:

1. *Лексический анализ*, в процессе которого считывается поток символов исходной программы, которые группируются в лексемы, представляющие собой определенные последовательности символов языка программирования, имеющие определенный смысл.
2. *Синтаксический анализ*. На этом этапе последовательность полученных лексем преобразуется в синтаксическое дерево разбора программы без учета семантики.
3. *Семантический анализ*. Здесь синтаксическое дерево переводится в семантическое, причем происходит проверка корректности исходной программы.
4. *Генерация кода*, после которой получается код на целевом языке, порожденный из промежуточного представления.

В этой работе будут продемонстрированы возможности расширения структуры компилятора. Под «расширением структуры» будем понимать введение новых нестандартных конструкций путем их внедрения в синтаксис языка. Основным методом для введения таких конструкций является «метод синтаксического сахара».

При расширении компилятора с помощью синтаксически сахарных конструкций появляется еще один шаг компиляции — desugaring — устранение синтаксического сахара. Этот процесс раскрытия синтаксически сахарных конструкций может быть произведен как на этапе синтаксического, так и на этапе семантического анализа. В настоящей работе будут реализованы сахарные конструкции на этапе синтаксиса с минимальными семантическими проверками.

Диссертация состоит из четырех глав. В первой главе «Синтаксический сахар» представлены основные понятия и примеры [гл. 1.1]. Также приведен анализ синтаксически сахарных конструкций в современных языках программирования [гл. 1.2].

Во второй главе «Реализация «сахара» в PascalABC.NET» сначала рассматриваются основные этапы компиляции PascalABC.NET [гл. 2.1]. Затем представлена информация о визиторах, которые напрямую использовались для внедрения синтаксически сахарных конструкций [гл. 2.2]. Далее описываются принципы введения синтаксического сахара [гл. 2.3] и проблемы семантических проверок для внедряемых конструкций [гл. 2.4].

Для внедрения «сахара» в синтаксис исходного языка, необходимо разобраться в структуре файлов, отвечающих за создание лексического и синтаксического анализаторов и внести изменения, соответствующие вводимым конструкциям [гл. 3.1]. Следующим шагом при внедрении сахара является этап десахаризации новой конструкции — desugaring [гл. 3.2]. После этого осуществляются соответствующие семантические проверки, необходимые для корректной работы внедренной конструкции [гл. 3.3].

В следующей главе «Введение оператора объединения со значением NULL— ??» по аналогии с предыдущей описываются последовательно этапы внедрения новой конструкции [гл. 4.1, 4.2]. Затем описывается встреченная

проблема повторяющихся вычислений и ее решение на этапе синтаксиса [гл. 4.3].

В заключительной пятой главе демонстрируются результаты работы внедренных синтаксически сахарных конструкций в виде примеров программ с результатами или сообщениями об ошибках, если программа заведомо некорректна.

Постановка задачи

1. Разработать методику реализации синтаксически сахарных расширений для компилятора PascalABC.NET
2. Применить разработанные методы для внедрения сахарных конструкций:
   - NULL-условного оператора (?.)
   - оператора объединения со значением NULL (??)
   - тип integer?
3. Встроить данные конструкции в грамматику языка, определив уровень приоритета и ассоциируемость
4. Решить проблему повторяющихся вычислений в секции операторов для ??

Глава 1. Синтаксический сахар

1.1 Основные понятия и примеры

Синтаксическим сахаром называется любой элемент синтаксиса, который может дать программисту альтернативный способ записи уже существующей в языке синтаксической конструкции. Но при этом такой способ является более удобным, или более кратким, или помогает писать программы в хорошем стиле.

«Сахар» выражается через конструкции базового языка и встраивается в структуру компилятора как на синтаксическом, так и на семантическом уровнях. Важно то, что синтаксический сахар можно удалить из языка программирования без потери его функциональных возможностей. Таким образом, синтаксический сахар предназначен лишь для того, чтобы сделать более комфортным процесс написания программы.

В качестве примера можно привести следующую языковую конструкцию — тернарную условную операцию ?:. Результат работы следующих двух фрагментов аналогичен:

```
if (a > 0) then
    b := c
else
    b := 0;
```

или

```
b := a > 0 ? c : 0;
```

Одной из причин введения такой конструкции стало желание вставить проверку простого условия непосредственно в выражении. Помимо этого, очевиден тот факт, что эта синтаксически сахарная конструкция действительно сокращает запись.

1.2 Сахарные конструкции в разных языках программирования

Вполне закономерным является то, что со временем из общей массы языков программирования начинают выделяться самые эффективные и продуманные языки, которые позволяют единовременно решать большое количество актуальных задач. На сегодняшний день в число таких языков входят в частности JAVA и C#. Синтаксис этих языков программирования в полной мере насыщен сахарными конструкциями.

Синтаксические возможности языка C# значительно расширяются за счет введения следующего «сахара»:

- Лямбда-функции
- Делегаты
- Перегрузка операторов
- Интерполяция строк
- Инициализация свойств со значением
- Поддержка yield в итерациях
- Поддержка оператора ??
- Поддержка Null-условного оператора (?.)

и многого другого.

Приведем несколько примеров внесенных изменений:

Использование лямбда-выражений:

Конструкция базового языка:

```
public string[] GetCountryList()
{
   return new string[] { "Russia", "USA", "UK" };
}
```

Синтаксический сахар:

```
public string[] GetCountryList() => new string[]
{ "Russia", "USA", "UK" };
```

Интерполяция строк:

Конструкция базового языка:

```
name = string.Format("Employee name is {0}, located
at {1}", emp.FirstName, emp.Location);
```

Синтаксический сахар:

```
name = $"Employee name is {emp.FirstName}, located at
{emp.Location}";
```

Null-условный оператор:

Конструкция базового языка:

```
string location = emp == null ? null : emp.Location;
string departmentName = emp == null ? null :
emp.Department == null ? null : emp.Department.Name;
```

Синтаксический сахар:

```
string location = emp?.Location;
string departmentName = emp?.Department?.Name;
```

В версии C# 6.0 имеется совершенно новый компилятор, написанный полностью на языке C#. Новый компилятор является модульным, поэтому в дополнение к компиляции исходного кода в исполняемый файл или библиотеку его функциональность можно задействовать многими другими путями. Известный под названием Roslyn [6], новый компилятор упрощает написание инструментов статического анализа, редакторов с подсветкой синтаксиса и автозавершением кода, а также подключаемых модулей Visual Studio, которые понимают код C#. Система Roslyn является наиболее идеологически близкой к компилятору PascalABC.NET.

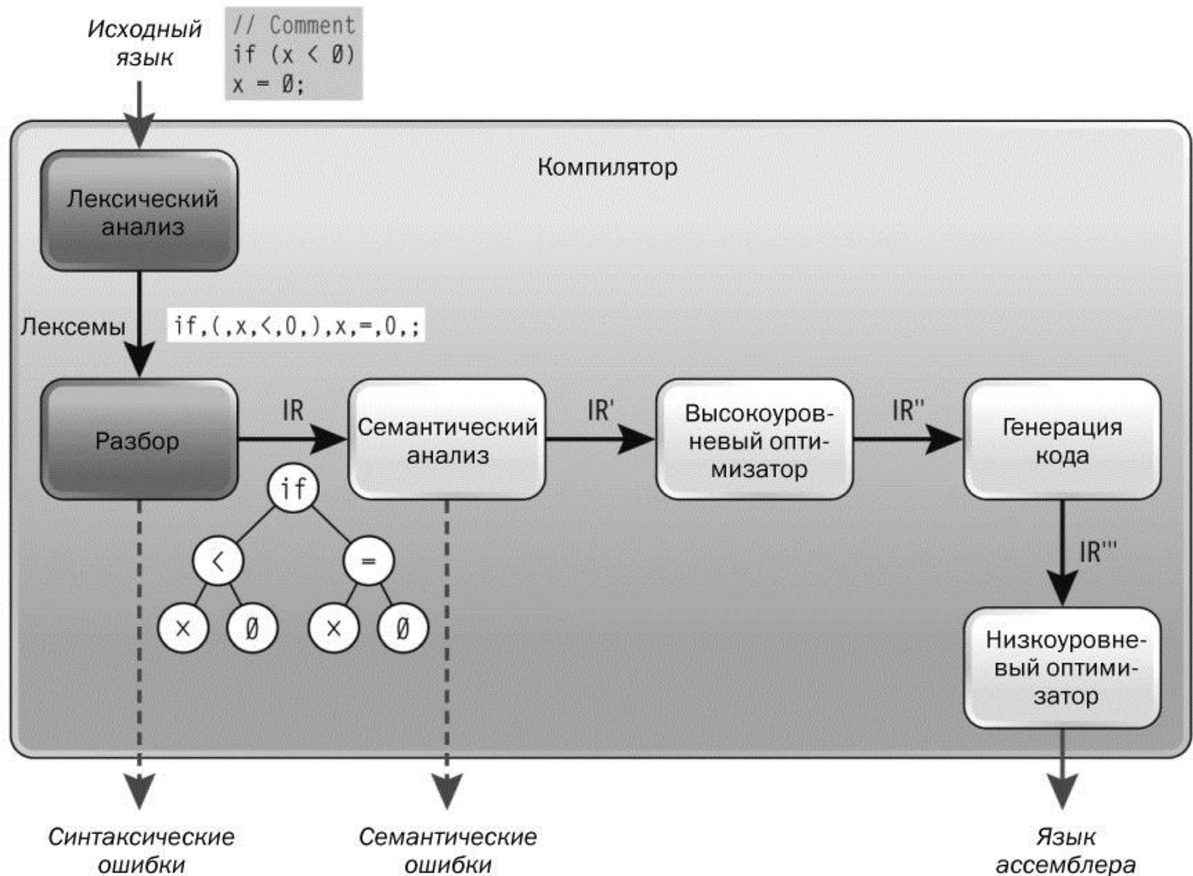


Рис 1. Основные этапы компилятора Roslyn

Другие наиболее известные исследования в области синтаксического сахара связаны с системой SugarJS [7] — open source (лицензия — MIT) библиотека для языка JavaScript. Она разработана, чтобы быть интуитивным и ненавязчивым инструментом, повышающим выразительность кода, который позволял бы делать больше с меньшим количеством строк и меньше задумываясь над рутиной. Система SugarJS производит очистку от сахарных конструкций на уровне исходного кода, после чего запускает базовый компилятор JavaScript, что менее эффективно, чем встраивание обработки

синтаксического сахара в компилятор. С одной стороны, такой подход обеспечивает некоторую гибкость, а с другой, приводит к проблемам с производительностью.

Изложенное в следующих главах опирается на расширение структуры компилятора языка программирования PascalABC.NET, который является реализацией языка Object Pascal для платформы Microsoft.NET.

Глава 2. Реализация «сахара» в PascalABC.NET

После множества внесенных модернизаций в структуру языка PascalABC.NET, он приблизился по своей выразительности и функциональности к языку C#. Помимо этого, он также пополнился такими конструкциями, как кортежи, срезы, последовательности, многие из которых реализовывались именно методом синтаксического сахара.

В данной главе будет в общих чертах рассмотрена структура компилятора PascalABC.NET, принцип расширения структуры компилятора синтаксически сахарными конструкциями и способы решения проблем, возникающих при реализации синтаксического сахара.

2.1 Общая структура компилятора PascalABC.NET

Рис. 2. Этапы компиляции в PascalABC.NET

Вначале исходный текст программы поступает на вход синтаксическому анализатору, который также называется парсером, и разбирается им в синтаксическое дерево.

Синтаксическое дерево содержит текст программы, представленный в виде дерева. В процессе построения такой структуры данных текст программы проверяется лишь на соответствие заданной формальной грамматике. Семантика на этом уровне практически не учитывается. Другими словами, синтаксическое дерево – это текст программы, который переведен парсером в представление, которое отражает синтаксическую структуру входной последовательности и хорошо подходит для дальнейшей обработки. Синтаксическое дерево не содержит никакой информации о сущностях, определенных в программе. Например, на синтаксическом уровне нет таблицы символов, содержащей данных обо всех переменных и их типах. Поэтому в процессе построения синтаксического дерева невозможно производить необходимые семантические проверки синтаксически сахарных узлов.

После того как синтаксическое дерево построено, на следующем этапе компиляции специальный визитор (конвертер деревьев) переводит его в семантическое дерево. На этом этапе происходит проверка корректности исходной программы (проверка типов и ряд других проверок).

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. Важно отметить, что на этапе построенного семантического дерева синтаксически сахарные узлы уже отсутствуют, они были представлены через базовые.

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.

2.2 Визиторы PascalABC.NET

Класс WalkingVisitorNew является потомком класса AbstractVisitor, который, в свою очередь, является реализацией интерфейса IVisitor. В классе AbstractVisitor представлена реализация методов 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.

В классе WalkingVisitorNew определен метод ReplaceUsingParent, реализующий алгоритм замены синтаксически сахарного узла на один из базовых:

```
public void ReplaceUsingParent(syntax_tree_node from,
syntax_tree_node to)
  {
      to.Parent = from.Parent;
      if (from.Parent == null)
          throw new Exception("У корневого элемента нельзя
получить Parent");
      from.Parent.ReplaceDescendant(from, to);
  }
```

Помимо метода ReplaceUsingParent(), в классе WalkingVisitorNew определен аналогичный метод замены сахарного узла только не на один десахарный, а на целую группу базовых — ReplaceStatementUsingParent().

Данный метод применяется исключительно для сахарных операторов, т.к. именно узел statement можно заменить на List<statement>. При такой замене несколько statement объединяются в операторные скобки begin-end. Реализация этого метода выглядит следующим образом:

```
public void ReplaceStatementUsingParent(statement from,
IEnumerable<statement> to, Desc d = Desc.DirectDescendants)
{
     foreach (var x in to)
          x.Parent = from.Parent;
     var sl = from.Parent as statement_list;
     if (sl != null)
     {
          sl.ReplaceInList(from, to);
     }
     else
     {
          var l = new statement_list();
          l.AddMany(to);
          l.source_context = from.source_context;
          from.Parent.ReplaceDescendant(from, l, d);
     }
}

public void ReplaceDescendant<T,T1>(T from, T1 to, Desc d =
Desc.All) where T: syntax_tree_node where T1 : T
          {
               var ind = FindIndex(from,d);
               this[ind] = to;
               to.Parent = from.Parent;
          }
public int FindIndex(syntax_tree_node node, Desc d =
Desc.All)
        {
            int ind = -1;
            var count = d == Desc.All ? subnodes_count :
          subnodes_without_list_elements_count;
            for (var i = 0; i < count; i++)
                if (node == this[i])
                {
                      ind = i;
                      break;
                }
            if (ind == -1)
```

```
                throw new Exception(string.Format("У элемента {0} не найден {1} среди дочерних\n", this, node));
            return ind;
        }
```

2.3 Принцип введения сахарных конструкций в PascalABC.NET

В процессе работы по внедрению сахарных конструкций в синтаксис языка PascalABC.NET были использованы описанные в [1] методы реализации синтаксически сахарных расширений в компиляторах. Далее рассмотрим изученные принципы расширений подробнее.

Введение синтаксически сахарных конструкций основывается в первую очередь на построении синтаксического поддерева для нового сахарного узла.

Пусть исходный язык представлен базовыми узлами синтаксического дерева с типами — $t_1 \ldots t_n$. После введения новых конструкций появляются сахарные синтаксические узлы с типами — $s_1 \ldots s_m$, которые в последствии необходимо устранить, выразив через базовые. Эта замена происходит на этапе перехода от синтаксического дерева к семантическому при помощи специального визитора (конвертера деревьев).

Специальный визитор состоит из множества методов visit(sug), которые меняют сахарные узлы $s_1 \ldots s_m$ на целые поддеревья, уже не содержащие сахар. Формально такой метод можно описать следующим образом:

```
visit(sug)
{
    Создание несахарного узла unsug;
    Замена sug на unsug;
    Посещение подузлов узла sug;
}
```

Одной из главных проблем, с которой можно столкнуться при реализации синтаксического сахара выше описанным методом, является

необходимость дополнительных проверок, возможных только на этапе семантики.

Рассмотрим несколько примеров синтаксически сахарных конструкций, встроенных в структуру компилятора PascalABC.NET:

- **Кортежи**

Конструкция базового языка: `Tuple.Create(a,b)`

Синтаксический сахар: `(a,b)`

В этом примере отсутствуют семантические проверки. Однако, существует ограничение на количество значений в кортеже — не более семи. Но эта проверка выполняется задолго до семантики, непосредственно перед созданием синтаксического узла.

- **Кортежное присваивание**

Конструкция базового языка:

```
begin
    var #c := c;
    a := #c.Item1;
    b := #c.Item2;
end;
```

Синтаксический сахар: `(a,b) := c`

В этом примере один оператор заменяется на целую последовательность операторов. Важно отметить, что для этой конструкции решена проблема повторяющихся вычислений путем введения уникальной переменной.

- **λ-выражения**

Лямбда-выражения представляют собой функции, создаваемые «на лету». Использование лямбда-выражений позволяет не описывать множество маленьких функций, которые используются в программе один раз. Они

облегчают написание и восприятие текста программы. Лямбда-выражения присутствуют практически во всех современных распространенных языках программирования.

Конструкция базового языка:

```
function f(x:integer): integer;
begin
    Result := x+1;
end;
```

Синтаксический сахар: `var f:=function(x:integer)->x+1`

При введении λ-выражений в структуру PascalABC.NET был реализован вывод типов формальных параметров и возвращаемого значения λ-выражений, а также реализован захват переменных из внешнего контекста.

- **Срезы вида для массивов, списков и строк**

Срез – это подмножество элементов коллекции в заданном диапазоне с заданным шагом

Срезы имеют вид `a[from : to]` или `a[from : to : step]`

Срезы реализованы для строк, динамических массивов и списков List. Срез для строки – это строка, для динамического массива – динамический массив, для списка – список

Хочется отметить, что кортежи и срезы появились в PascalABC.NET раньше, чем в C#. И они позволяют манипулировать при решении задач новыми высокоуровневыми сущностями, улучшая стиль решения.

2.4 Проблема семантических проверок

Рассмотрим пример:

`a++ → a+=1;`

Реализация постфиксного инкремента a++ (отсутствующая в базовом языке) осуществляется заменой на оператор присваивания со сложением (a+=1), имеющийся в базовом языке. При такой не эквивалентной замене может возникнуть ошибка, так как оператор присваивания со сложением допускает сложение вещественных чисел, а оператор инкремента обязан работать только для целочисленных. Чтобы не допустить такую ошибку, необходимо выполнить соответствующую проверку, которая может быть реализована двумя способами:

Способ 1. Проверка в визиторе преобразования в семантическое дерево

Сделать семантическую проверку перед заменой сахарного узла на поддерево базовых узлов в соответствующем методе visit(), который формально описывается следующим образом:

```
visit (sug:inc_node)
{
    IsIntType(sug.var.type); //если не целый, то ошибка
    var unsug = assignplus(sug.var, int_const(1));
    Замена sug на unsug;
    visit (unsug);
}
```

Пример:

Конструкция базового языка:

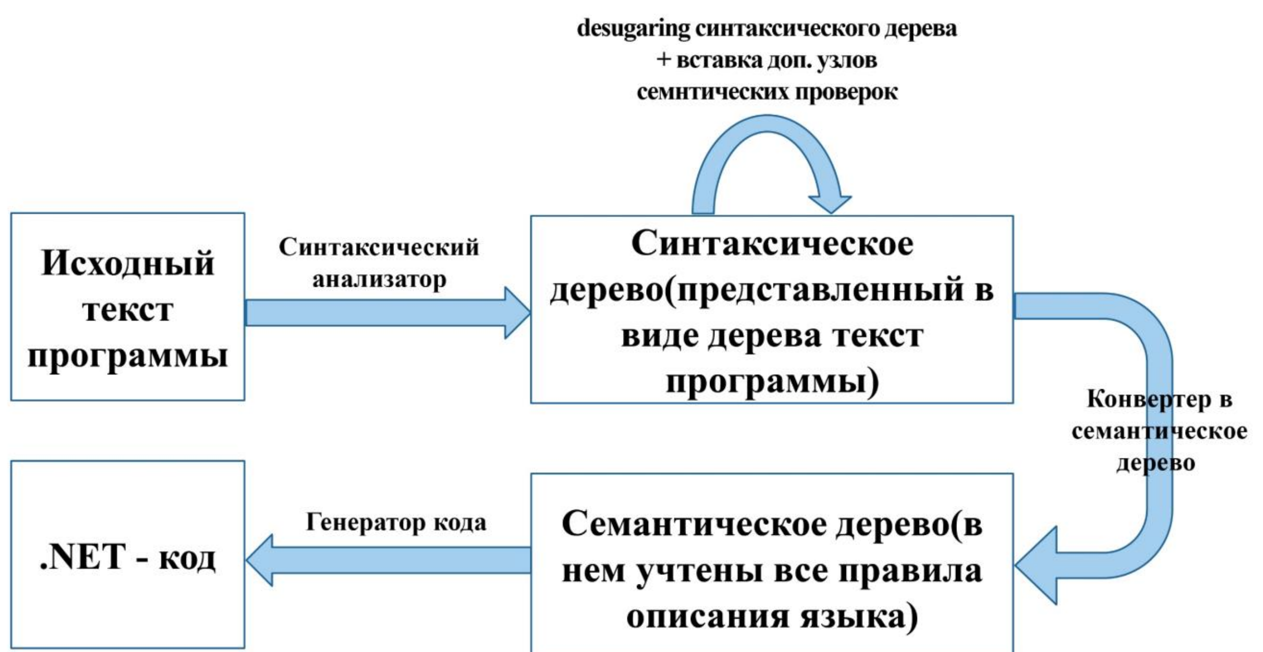
```
var tup: Tuple<integer, string>
```

Синтаксический сахар:

```
var tup: (integer, string)
```

На синтаксическом уровне трудно отличить кортеж от перечислимого типа, поэтому используется первый способ проверки непосредственно в визиторе преобразования в семантическое дерево

Способ 2. Проверка в специальном узле синтаксического дерева



Данный способ реализуется в два последовательных этапа. На первом этапе специальный визитор производит десахаризацию, заменяя сахарный узел на поддерево, содержащее базовые узлы и специальный проверочный:

```
visit (sug:inc_node)
{
    var check = sem_check_statement(sug);
    var unsug = assignplus(sug.var, int_const(1));
    Замена sug на (check, unsug);
    visit (unsug);
}
```

На втором этапе визитор преобразования в семантическое дерево, при обходе узла `sem_check_statement` выполнит необходимую семантическую проверку:

```
visit (check: sem_check_statement)
{
    If (check.statement is inc_node)
        IsIntType((check as inc_node).var.type);
    //если не целый, то ошибка
}
```

Пример:

Конструкция базового языка:

```
var b := a.SystemSlice(1,5,2);
```

Синтаксический сахар:

```
var b := a[1,5,2]
```

В этом примере используется второй способ: desugaring осуществляется в визиторе, преобразующем синтаксическое дерево. Наиболее чистый способ, т.к. синтаксические действия не переносятся на семантический уровень.

Глава 3. Введение NULL-условного оператора — ?.

3.1 Изменение грамматики языка

Для того, чтобы внедрить новую сахарную конструкцию в синтаксис языка PascalABC.NET, в первую очередь необходимо внести изменения в грамматику языка.

Для генерации лексического анализатора – сканера и синтаксического – парсера, необходимо два файла соответственно: один с расширением. lex, второй с расширением. yacc. Познакомимся с их структурой.

Структура .lex файла выглядит следующим образом:

```
Описания
%%
Правила
%%
Пользовательский код
```

Раздел описаний содержит определения макросимволов (метасимволов) вида {имя} {выражение}. Если в последующем тексте в регулярном выражении встречается {имя}, то оно заменяется на {выражение}. Если строка описаний начинается с пробелов или заключена в скобки %{...}%, то она просто копируется в выходной файл.

Раздел правил содержит строки вида {выражение} {действие}. Действие – это фрагмент программы на C#, который выполняется тогда, когда обнаружена цепочка, соответствующая {выражению}.

Раздел программ может содержать произвольные тексты на C# и целиком копируется в выходной файл.

Структура файла.yacc аналогична структуре файла.lex. Она состоит из трех секций – секции описаний, секции правил, в которой описывается грамматика, и секции программ. Пример простейшей программы на YACC'е:

```
%token name
 %start e
 %%
 e : e '+' m
   | e '-' m
   | m ;
 m : m '*' t
   | m '/' t
   | t ;
 t : name
   | '(' e ')';
     %%
```

Секция правил содержит информацию о том, что символ name является лексемой (терминалом) грамматики, а символ e – ее начальным нетерминалом. Символы : | ; – принадлежат метаязыку и читаются "есть по определению", "или" и "конец правила" соответственно. С каждым правилом грамматики может быть связано действие, которое будет выполнено при свертке по данному правилу. Оно записывается в виде заключенной в фигурные скобки последовательности операторов языка Си, расположенной после соответствующего правила.

Для введения NULL-условного оператора ?. необходимо добавить новый терминальный символ в lex-файле:

```
"?."         { return (int)Tokens.tkQuestionPoint; }
```

Далее в первом разделе yacc-файла описываются терминальные и нетерминальные символы:

```
%token <ti>  tkQuestionPoint
%type <ex> var_question_point
```

Затем в следующем разделе yacc-файла определяются сами правила языка. Для этого используется формальный способ описания, с использованием форм Бэкуса – Наура (БНФ) – система описания синтаксиса, в которой одни синтаксические категории описываются через другие, а

именно: нетерминальные символы описываются через терминалы/лексемы/токены:

```
var_reference
     : var_address variable
     {
          $$ = NewVarReference($1 as get_address, $2
               as addressed_value, @$);
     }
     | variable
       { $$ = $1; }
     | var_question_point
       { $$ = $1; }
     ;

var_question_point
     : variable tkQuestionPoint variable
     {
          $$ = new dot_question_node($1 as addressed_value,
               $3 as addressed_value, null, @$);
     }
     | variable tkQuestionPoint var_question_point
     {
          $$ = new dot_question_node($1 as addressed_value,
               $3 as addressed_value, null, @$);
     }
     ;
```

Именно на этом этапе важно помнить о приоритете вводимых операций по отношению к уже существующим. NULL-условный оператор ?. имеет наивысший приоритет, поэтому в секции грамматических правил он вводится на уровне с оператором @. Далее приведена таблица приоритетов базовых операций:

@, not, ^, +, - (унарные), new	1 (наивысший)
***, /, div, mod, and, shl, shr, as, is**	2
+, - (бинарные), or, xor	3
=, <>, <, >, <=, >=, in, =>	4 (низший)

На этом изменение .yacc и .lex файлов закончено. По введенным грамматическим правилам будет создаваться соответствующий сахарный узел, который в последствии на этапе перехода от синтаксического дерева к семантическому будет заменяться на базовый.

Следующий шаг – это генерация узла синтаксического дерева для сахарной конструкции — `dot_question_node`.

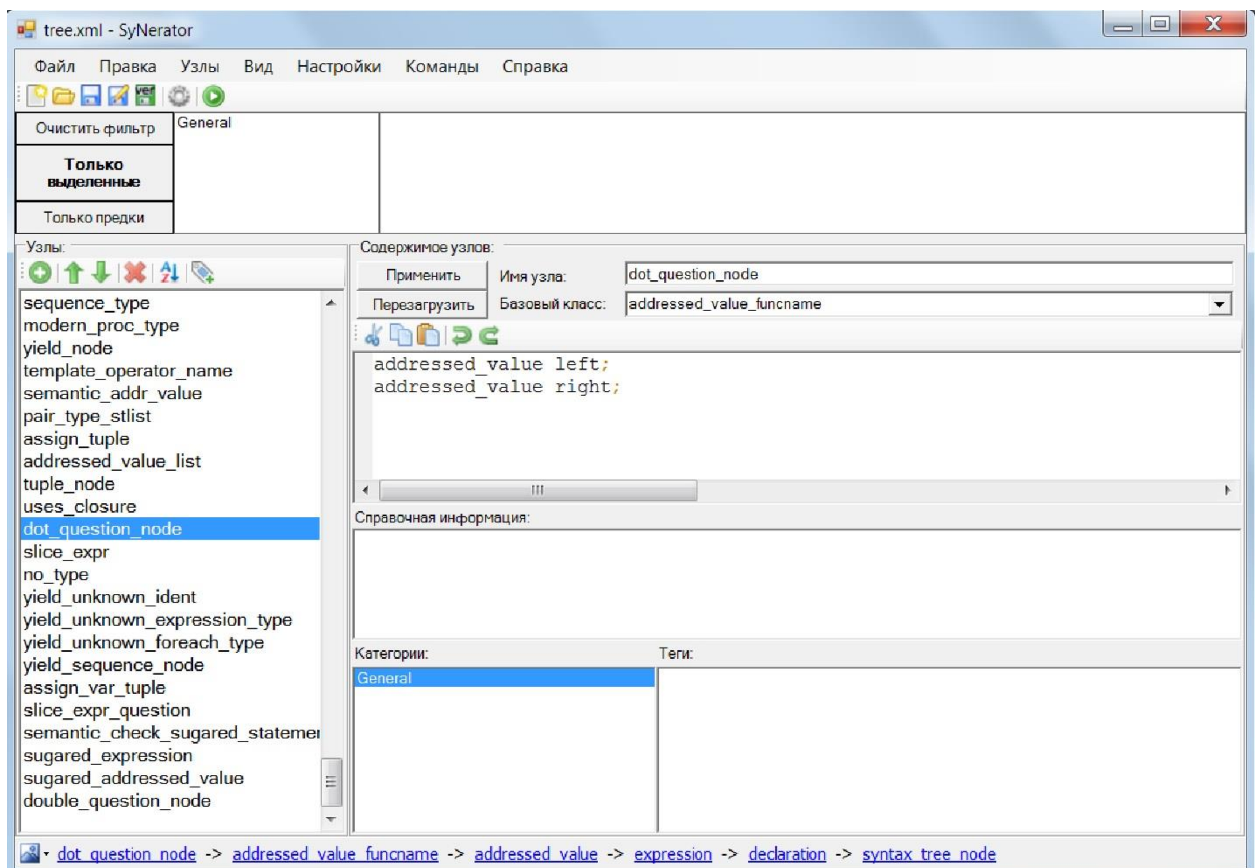


Рис. 3. Генератор узлов синтаксического дерева `NodesGenerator`

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. Генератор узлов и ряда визиторов имеет внешний вид, изображенный на рис. 3.

С помощью этой утилиты генерируем новый узел с именем dot_question_node. Базовым классом для этотого узла является addressed_value_fucnname. В теле определяем подузлы, соответствующие левому и правому операндам:

```
addressed_value left;
addressed_value right;
```

На этом этапе компилятор уже может распознать вводимую конструкцию и поставить ей в соответствие узел double_question_node в синтаксическом дереве программы.

Далее будет описан алгоритм обработки и замены сахарного узла dot_question_node в синтаксическом дереве программы, а также необходимые семантические проверки.

3.2 DESUGARING

Реализация null-условного оператора ?. (отсутствующего в базовом языке) осуществляется заменой на тернарную условную операцию ?: :

a?.b → a=nil?nil:a.b

Для этого создается специальный класс `QuestionPointDesugarVisitor.cs`, в котором переопределяется метод `visit()` для синтаксически сахарного узла `dot_question_node`. На этом этапе пока отсутствуют какие-либо семантические прповерки.

Основная сложность, которая была встречена при реализации алгоритма десахаризации оператора ?., заключалась в обеспечении корректности ассоциирования. Эта проблема оказалась полной неожиданностью при реализации сахарной конструкции. И она была решена на этапе построения синтаксического дерева с помощью реализации рекурсивного алгоритма «протаскивания».

Остановимся подробнее на проблематике и основной идее алгоритма. Для этого рассмотрим пример:

```
a?.b?.c
```

Такое выражение ошибочно будет разобрано так, как формально показано на рис. 4б:

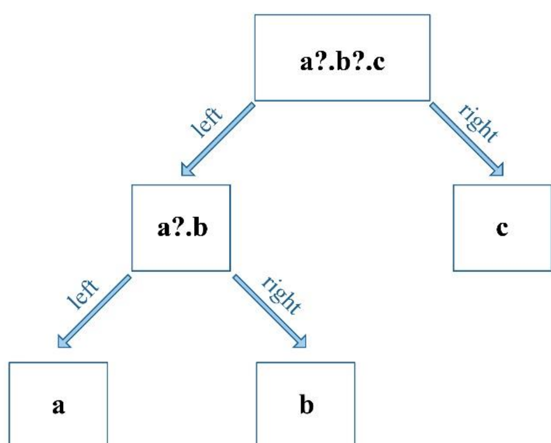


Рис 4а. Корректный разбор

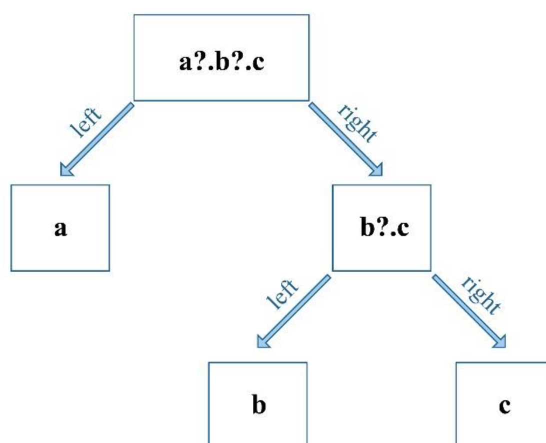
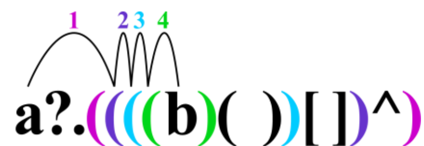


Рис 4б. Ошибочный разбор

Помимо повторения оператора ?., грамматика допускает более сложные конструкции:

a?.b( )[ ]^

В этом случае `left = a, right = b( )[ ]^`. И задача состоит в том, чтобы через все эти конструкции «протащить» левый операнд оператора ?. к правому операнду:



Для решения этой проблемы был реализован метод `Into()`, который обрабатывает всевозможные случаи и обеспечивает корректность при ассоциировании, как показано на рис. 4а. Алгоритм `Into()` выглядит следующим образом:

```
   public addressed_value Into(addressed_value x,
addressed_value v)
           {
               if (v.GetType() ==
typeof(dot_question_node))
               {
                   var vv = v as dot_question_node;
                   var res = new dot_question_node(Into(x,
vv.left), vv.right, x.source_context);
                   res.left.Parent = res;
                   res.right.Parent = res;
                   return res;
               }
               else if (v.GetType() == typeof(dot_node))
               {
                   var vv = v as dot_node;
                   var res = new dot_node(Into(x, vv.left),
vv.right, x.source_context);
```

```
                    res.left.Parent = res;
                    res.right.Parent = res;
                    return res;
                }
                else if (v.GetType() == typeof(indexer))
                {
                    var vv = v as indexer;
                    var res = new indexer(Into(x,
vv.dereferencing_value), vv.indexes, x.source_context);
                    res.dereferencing_value.Parent = res;
                    res.indexes.Parent = res;
                    return res;
                }
                else if (v.GetType() == typeof(slice_expr))
                {
                    var vv = v as slice_expr;
                    var res = new slice_expr(Into(x,
vv.dereferencing_value), vv.from, vv.to, vv.step,
x.source_context);
                    res.dereferencing_value.Parent = res;
                    res.from.Parent = res;
                    res.to.Parent = res;
                    res.step.Parent = res;
                    return res;
                }
                else if (v.GetType() ==
typeof(slice_expr_question))
                {
                    var vv = v as slice_expr_question;
                    var res = new
slice_expr_question(Into(x, vv.dereferencing_value),
vv.from, vv.to, vv.step, x.source_context);
                    res.dereferencing_value.Parent = res;
                    res.from.Parent = res;
                    res.to.Parent = res;
                    res.step.Parent = res;
                    return res;
                }
```

```
                else if (v.GetType() == typeof(method_call))
                {
                    var vv = v as method_call;
                    var res = new method_call(Into(x,
vv.dereferencing_value), vv.parameters, x.source_context);
                    res.dereferencing_value.Parent = res;
                    res.parameters.Parent = res;
                    return res;
                }
                else if (v.GetType() ==
typeof(roof_dereference))
                {
                    var vv = v as roof_dereference;
                    var res = new roof_dereference(Into(x,
vv.dereferencing_value), x.source_context);
                    res.dereferencing_value.Parent = res;
                    return res;
                }
                else if (v.GetType() ==
typeof(ident_with_templateparams))
                {
                    var vv = v as ident_with_templateparams;
                    var res = new
ident_with_templateparams(Into(x, vv.name),
vv.template_params, x.source_context);
                    res.name.Parent = res;
                    res.template_params.Parent = res;
                    return res;
                }
                else
                {
                    var res = new dot_node(x, v,
x.source_context);
                    res.left.Parent = res;
                    res.right.Parent = res;
                    return res;
                }
            }
```

В реализованном подходе необходимо учитывать следующую особенность: при появлении новых конструкций в грамматике `variable`, нужно добавить в алгоритм `Into()` ветку, обрабатывающую новую конструкцию. Дальнейшие разработки будут направлены на поиск более универсального алгоритма, не требующего дополнительных изменений при появлении новых синтаксических возможностей языка PascalABC.NET.

3.3 Семантические проверки

На этом этапе важно учитывать тот факт, что узел выражения в синтаксическом дереве в отличие от узла оператора может быть заменен только на один узел, поэтому дополнительные семантические действия выполняются следующим образом.

Сахарный узел `dot_question_node` заменяется на специальный узел `sugared_addressed_value`, который хранит как компоненты сахарного узла, так и заменяющий узел.

```
public override void visit(dot_question_node dqn)
{
     var qce = ConvertToQCE(dqn); // преобразование ?. к ?:
     var sug = sugared_addressed_value.NewP(dqn, qce,
dqn.source_context);
     ReplaceUsingParent(dqn, sug);
     visit(qce);
}
```

При обходе узла `sugared_addressed_value` семантическим визитором осуществляется семантическая проверка компонент сахарного узла, после чего обходится заменивший его узел для преобразования вложенных в него конструкций.

В данном случае необходимо осуществить следующие семантические проверки:

1. Левый операнд имеет ссылочный тип
2. Если правый операнд имеет размерный тип T, то преобразовать его к типу Nullable <T>. Отметим некоторые особенности, связанные с типом Nullable <T>: он является размерным, но переменным такого типа можно присвоить nil (сгенерируется специальный код) и сравнивать с nil в отличие от переменных любого другого размерного типа.

Первая проверка реализуется в классе semantic_checks_for_sugar для уже десахарного узла question_colon_expression:

```
public void
semantic_check_dot_question(SyntaxTree.question_colon_expre
ssion qce)
{
     var av = convert_strong((qce.condition as
bin_expr).left);
     if (!av.type.is_class)
        AddError(av.location,
"OPERATOR_DQ_MUST_BE_USED_WITH_A_REFERENCE_TYPE");
}
```

Затем этот метод вызывается при переопределении visit() для sugared_addressed_value, в котором осуществляется вторая семантическая проверка и преобразование к типу Nullable <T>:

```
public override void
visit(SyntaxTree.sugared_addressed_value av)
{
     ...

     else if (av.sugared_expr is
SyntaxTree.dot_question_node)
     {
          var qce = av.new_addr_value as
     SyntaxTree.question_colon_expression;
          var av_cs = convert_strong(qce.ret_if_false);
          if (!type_table.is_with_nil_allowed(av_cs.type))
          {
               var dn =
               new dot_node(new ident("PABCSystem"),new
               ident("DQNToNullable"));
               (av.new_addr_value as
               SyntaxTree.question_colon_expression).ret_if_f
               alse =
               new method_call(dn, new
               expression_list((av.new_addr_value as
               SyntaxTree.question_colon_expression).ret_if_f
               alse), av.source_context);
                  }

semantic_check_dot_question(av.new_addr_value as
SyntaxTree.question_colon_expression);
          }
          ...
          ProcessNode(av.new_addr_value); // обойти
     десахарное
}
```

В связи с тем, что мы не хотим на этом этапе явно писать тип T, в модуле PABCSystem.pas реализуется метод для привидения переменной размерного типа T к Nullable<T> — DQNToNullable<T>(). А при посещении узла sugared_addressed_value происходит вызов этого метода для правого операнда, в случае если он окажется размерного типа.

Глава 4. Введение оператора объединения со значением NULL — ??

Данная глава посвящена расширению структуры компилятора PascalABC.NET, путем внедрения новой синтаксически сахарной конструкции — оператора ??, а также решению проблемы повторных вычислений в реализации оператора.

4.1 Изменение грамматики языка

Первым делом для введения новой конструкции необходимо внести соответствующие изменения в файлы, отвечающие за грамматику языка. В lex-файле определяем новый терминальный символ, именуемый токеном, не вызывающий конфликтов с введенными ранее:

```
"??"      { return (int)Tokens.tkDoubleQuestion; }
```

Далее в первом разделе yacc-файла описываем этот токен и нетерминал, соответствующий новой конструкции:

```
%token <ti> tkDoubleQuestion
%type <ex> double_question_expr
```

На следующем шаге, прежде чем определить новые грамматические правила, необходимо определить уровень приоритета для оператора ??. Было принято решение, по аналогии с языком программирования C#, отнести данную конструкцию к классу с самым низким приоритетом, и определить для нее на одном уровне с оператором ?: следующие грамматические правила:

```
expr_l1
    : double_question_expr
        { $$ = $1; }

    | question_expr
        { $$ = $1; }
    ;

double_question_expr
    : relop_expr
        { $$ = $1; }
    | double_question_expr tkDoubleQuestion relop_expr
        { $$ = new double_question_node($1 as
expression, $3 as expression, @$);}
```

По этим грамматическим правилам будет создаваться соответствующий сахарный узел, который в последствии на этапе перехода от синтаксического дерева к семантическому будет заменяться на базовый. С помощью утилиты `NodesGenerator` генерируем новый узел с именем `double_question_node`. Базовым классом для этотого узла является `addressed_value_fucnname`. В теле определяем подузлы, соответствующие левому и правому операндам:

```
expression left;
expression right;
```

На этом этапе компилятор уже может распознать вводимую конструкцию и поставить ей в соответствие узел `double_question_node` в синтаксическом дереве программы. Дальнейшая обработка будет происходить на этапе семантического анализа.

В следующих разделах будет описан алгоритм обработки и замены в синтаксическом дереве сахарного узла double_question_node на базовый, а также проблема повторяющихся вычислений и ее решение для блока операторов.

4.2 DESUGARING

Реализация оператора ?? (отсутствующего в базовом языке) осуществляется заменой на тернарную условную операцию ?: :

a??b → a=nil?b:a

Для этого создается специальный класс — DoubleQuestionDesugarVisitor.cs, в котором переопределяется метод visit() для синтаксически сахарного узла double_question_node.

```
public override void visit(double_question_node dqn)
{
            expression left = dqn.left;
            expression right = dqn.right;
            var eq = new bin_expr(new ident(tname), new
nil_const(), Operators.Equal, dleft.source_context);
            var qce = new question_colon_expression(eq,
right, left, dqn.source_context);
            ReplaceUsingParent(dqn, qce);
            visit(qce);
        }
```

4.3 Проблема повторяющихся вычислений

В процессе тестирования реализованного подхода была выявлена проблема с повторяющимися вычислениями в левом операнде. Такой повтор связан с тем, что при десахаризации узел double_question_node

заменяется на узел `question_colon_expression`, в котором левый операнд упоминается дважды. Схематично эта проблема представлена на рис. 5:

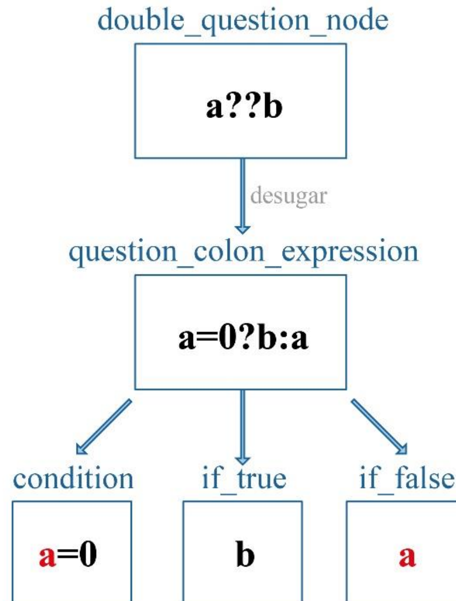


Рис. 5

Для решения этой проблемы в секции операторов достаточным будет ввести новую переменную до выполнения оператора. Например, конструкцию следующего вида:

```
var v:=a??b;
```

необходимо заменить на:

```
var tmp:=a;
var v:=tmp=nil?b:tmp;
```

Для осуществления представленного алгоритма в блоке операторов необходимо подниматься по синтаксическому дереву, обращаясь каждый раз к полю Parent, пока не будет встречен statement. Затем заменить в синтаксическом дереве этот оператор на последовательность операторов, при

этом генерируется имя новой переменной, которое не должно совпадать с уже существующими именами. Таким образом метод `visit()` для `double_question_node` переопределяется следующим образом:

```
public override void visit(double_question_node dqn)
        {
            var st = dqn.Parent;
            while (!(st is statement))
                st = st.Parent;
            var tname = "#temp" + UniqueNumStr();
            var tt = new var_statement(new ident(tname),
dqn.left);
            tt.var_def.Parent = tt;
            var l = new List<statement>();
            l.Add(tt);
            l.Add(st as statement);
            expression right = dqn.right;
            var eq = new bin_expr(new ident(tname), new
nil_const(), Operators.Equal, dqn.left.source_context);
            var qce = new question_colon_expression(eq,
right, new ident(tname), dqn.source_context);
            ReplaceUsingParent(dqn, qce);
            visit(qce);
            ReplaceStatementUsingParent(st as statement,
l);
            visit(tt);
        }
```

Глава 5. Тестовые программы и их результаты

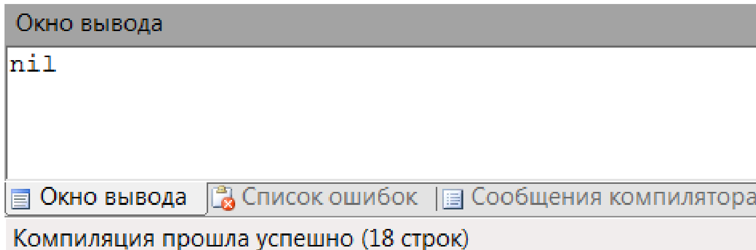
Пример 1. Иллюстрация корректности ассоциирования

1.1 Случай a?.b?.c

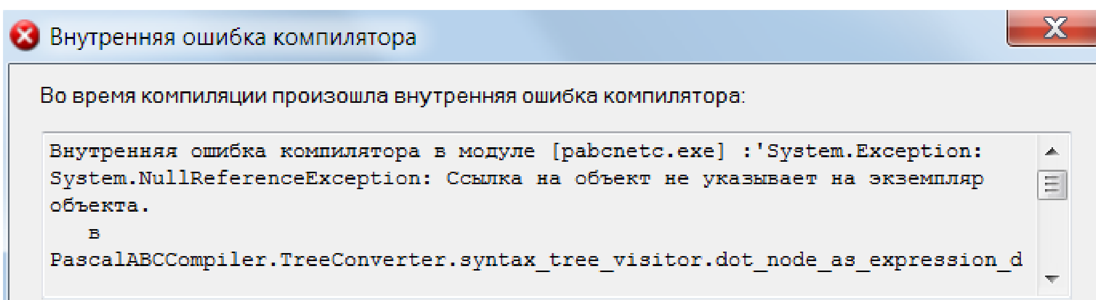
```
type
  Class1 = class
  public
    c: integer;
  end;

type
  Class2 = class
  public
    b: Class1;
  end;

begin
  var a := new Class2;
  var d := a?.b?.c; //в этом случае a.b = nil, поэтому d := nil
  print(d);
end.
```

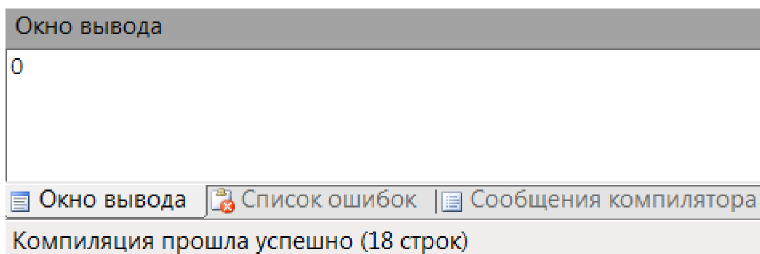


Если закомментировать в алгоритме `Into()` участок кода, отвечающий за обработку `?.`, то при попытке откомпилировать эту программу возникает внутренняя ошибка компилятора:



Если инициализировать `a.b`, то результат работы программы будет несколько другим:

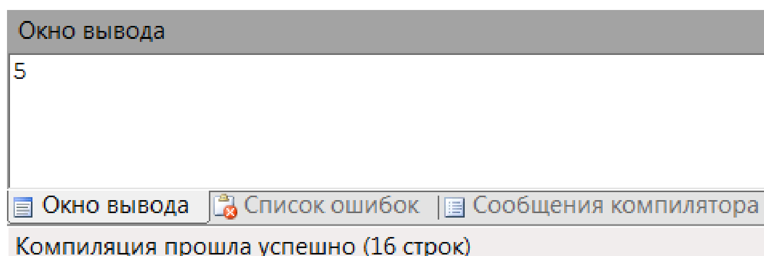
```
begin
  var a := new Class2;
  a.b := new Class1; //в этом случае a.b <> nil
  var d := a?.b?.c;  //поэтому d := a.b.c
  print(d);
end.
```



1.2 Случай a?.b[ ]

```
type
  Class1 = class
  public
    b: array[1..9] of integer;
    constructor (c: set of integer);
    begin
      foreach var i in c do
        b[i]:=i;
    end;
  end;

begin
  var a := new Class1([1,2,3,4,5,6,7,8,9]);
  var c := a?.b[5];
  print(c);
end.
```

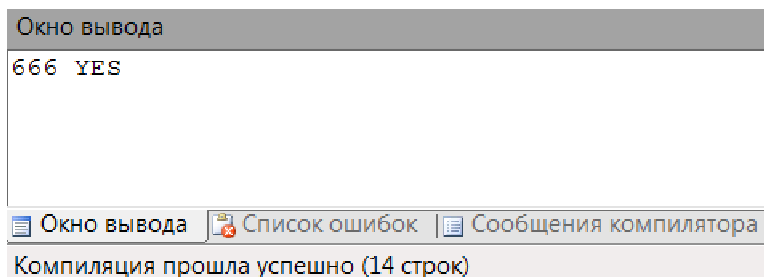


Аналогично предыдущему примеру: если закомментировать в алгоритме `Into()` участок кода, отвечающий за обработку `[]`, то при попытке откомпилировать эту программу возникает внутренняя ошибка компилятора.

1.3 Случай a?.b( )

```
type Class1 = class
public
  function b(x:string): string;
  begin
    Result := x;
    Print(666);
  end;
end;

begin
  var a := new Class1;
  var v := a?.b('YES');
  write(v);
end.
```

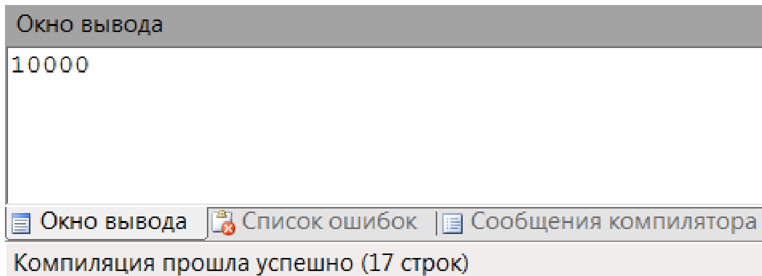


1.4 Случай a?.b[ ]^

```
type
 arrayOfinteger =  array[1..2] of ^integer;


type
  Class1 = class
  public
    b: arrayOfinteger;
end;
```

```
begin
  var a := new Class1;
  var c :integer := 10000;
  a.b[2] := @c;
  var d := a?.b[2]^;
  print(d);
end.
```

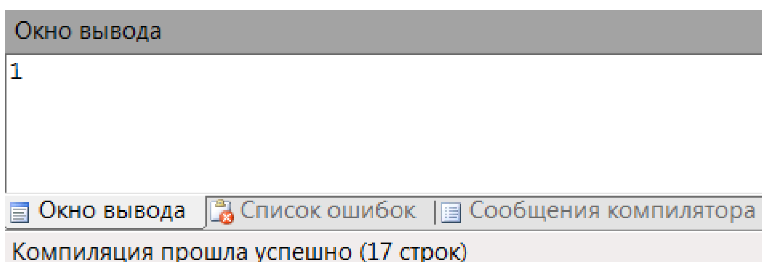


1.4 Случай a?.b([,])[ ]^

```
type
 arrayOfinteger = array[1..2] of ^integer;

type Class1 = class
public
  function b(x: set of integer): arrayOfinteger;
  begin
    foreach var i in x do
        Result[i]:= @i;
  end;
end;

begin
  var a := new Class1;
  var c := a?.b([1,2])[1]^;
  print(c);
end.
```

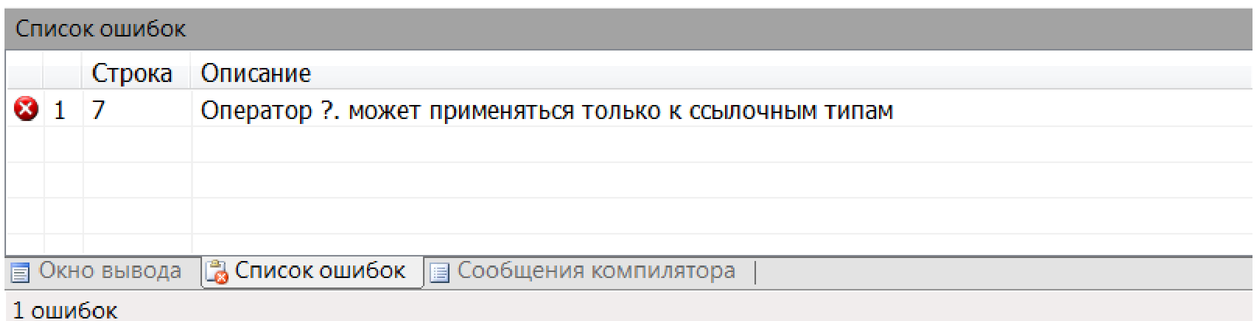


Пример 2

Иллюстрация корректности вывода ошибки, возникшей на этапе построения синтаксического дерева. Для этого попытаемся применить оператор ?. к переменной не ссылочного типа:

```
type A = record
  x: string;
end;

begin
  var a1 := new A;
  var v := a1?.x;
  write(v);
end.
```



Пример 3

Иллюстрация корректной работы в случае, когда правый операнд размерного типа. Корректность достигается за счет приведения к типу Nullable <T>:

```
type A = class
  x: integer;
end;

begin
  var a1 := new A;
  var v :=a1?.x;
  write(v);
end.
```

Окно вывода

```
0
```

Окно вывода | Список ошибок | Сообщения компилятора

Компиляция прошла успешно (9 строк)

Пример 4

Иллюстрация корректности выполнения внедренных в синтаксис языка операторов с учетом их приоритета относительно друг друга. А также следующий пример показывает, что решена проблема двойных вычислений в секции операторов :

```
type A = class
  x: string;
end;

function ss: string;
begin
 Result := '1';
 Print(666);
end;

begin
  var a1 := new A;
  a1.x :='2';
  var s : string := '3';
  var v := ss??a1?.x??s;
  write(v);
end.
```

Окно вывода

```
666 1
```

Окно вывода | Список ошибок | Сообщения компилятора

Компиляция прошла успешно (17 строк)

Заключение

В результате проделанной работы была расширена структура компилятора PascalABC.NET. Были решены следующие задачи:

1. Разработать методику реализации синтаксически сахарных расширений для компилятора PascalABC.NET
2. Применить разработанные методы для внедрения сахарных конструкций:
    a. NULL-условного оператора (?.)
    b. оператора объединения со значением NULL (??)
    c. тип integer?
3. Встроить данные конструкции в грамматику языка, определив уровень приоритета и ассоциируемость
4. Решить проблему повторяющихся вычислений в секции операторов для ??

В работе продемонстрировано то, что синтаксически сахарные конструкции могут быть раскрыты преимущественно на синтаксическом уровне с минимальными семантическими действиями. Также были определены действия, которые необходимо выполнять на семантическом уровне, по возможности минимизировано количество таких действий и локализованы места их вызова.

В ходе работы также был проделан разбор инфраструктуры проекта PascalABC.NET, для возможности внедрения. Все поставленные цели были достигнуты.

Литература

1. Михалкович С.С. Проблемы реализации синтаксически сахарных конструкций в компиляторах. Языки программирования и компиляторы — 2017: труды конференции / Ростов-на-Дону : Издательство ЮФУ, 2017, с. 209-212. Сайт проекта PascalABC.NET — http://pascalabc.net
2. Ахо А.В., Сети Р., Ульман Д.Д. – Компиляторы: принципы, технологии и инструменты
3. Сайт справочных материалов корпорации Microsoft — https://msdn.microsoft.com
4. Д. Албахари, Б. Албахари. C# 5.0. Справочник. Полное описание языка. М.: Вильямс, 2014.
5. The .NET Compiler Platform «Roslyn»; URL: https://github.com/dotnet/roslyn
6. Сайт библиотеки SugarJS для Javascript — https://sugarjs.com