МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное учреждение высшего образования «Южный федеральный университет»

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

Алгасов Александр Сергеевич

РЕАЛИЗАЦИЯ PATTERN MATCHING ДЛЯ PASCALABC.NET

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
по направлению 02.04.02 — Фундаментальная информатика и информационные технологии

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

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

Ростов-на-Дону — 2018

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Языки программирования являются неотъемлемой частью современных информационных технологий. Они являются основным инструментом создания программного обеспечения. На сегодняшний день различные виды программного обеспечения используются во множестве различных сфер и для решения различных задач. Среди них помощь в обучении и изучении нового материала, решение объемных вычислительных задач, оптимизация уже существующих. Программное обеспечение также упрощает и позволяет автоматизировать множество задач, которые ранее приходилось решать вручную. Примером могут служить генераторы компиляторов [1]. Кроме того, с каждым днем сложность задач, которые нельзя решить без применения компьютерных технологий и программного обеспечения растет. В свою очередь, это требует развития языков программирования и компиляторов.

На данный момент существует множество различных языков программирования. Они имеют различные парадигмы, возможности и особенности, однако, активно развивающиеся на сегодняшний день языки программирования часто влияют друг друга. В частности, сопоставление с образцом было более распространено в функциональных языках программирования. Однако, с появлением и развитием таких языков как Swift [2] и C#, сопоставление с образцом стало популярно и в объектно-ориентированных языках. В языках программирования сопоставление с образцом — это проверка выражения на соответствие некоторому шаблону. Часто встречающаяся задача — это выбор необходимого действия в зависимости от входных данных. Ее

можно решать по-разному, одним из способов является сопоставление входных данных нескольким шаблонам и выбор следующего шага алгоритма в зависимости от того, какому из шаблонов соответствует проверяемое выражение. Такие шаблоны могут иметь древовидную структуру, что позволяет осуществлять проверку на основе глубокого анализа структуры объекта.

В данной работе описывается реализация сопоставления с образцом в языке программирования PascalABC.NET [3]. Данный продукт является промышленным языком Pascal нового поколения и активно используется совместно с задачником Programming Taskbook в целях обучения языкам программирования [4]. Проведенное исследование относится к задачам компиляторостроения. В процессе исследования были изучены подобные языковые возможности в других программных продуктах. Большое количество информации было получено при изучении открытого набора компиляторов платформы .NET «Roslyn» [5]. В частности изучение компилятора C# позволило узнать особенности реализации сопоставления с образцом для языка программирования на той же платформе.

В процессе исследования и реализации были использованы средства системы контроля версий GitHub [6]. Системы контроля версий являются неотъемлемой частью разработки командных проектов. Компилятор PascalABC.NET и его среда разработки находятся как в этой системе как программный продукт с открытым исходным кодом и распространяется на условиях GNU LGPL версии 3 [7]. Такая лицензия используется для продуктов, которые состоят как из свободных для изменения, открытых частей, так и проприетарного программного обеспечения.

Разработка велась с использованием свободной Community версии встроенной среды разработки Visual Studio 2017 [8].

ПОСТАНОВКА ЗАДАЧИ

Реализовать в системе PascalABC.NET следующие возможности сопоставления с образцом:

- паттерн проверки типа выражения;
- паттерн деконструкции — возможность деконструировать объект на его составляющие;
- рекурсивный паттерн — возможность использовать паттерны вместо выходных переменных;
- возможность описывать способ деконструкции объекта на его составляющие;
- операцию позволяющую проверить выражение на соответствие паттерну;
- оператор выбора необходимого действия в зависимости от соответствия выражения одному из паттернов.

ГЛАВА 1

СОПОСТАВЛЕНИЕ С ОБРАЗЦОМ В ДРУГИХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

В этой главе рассматриваются возможности сопоставления с образцом в других языках программирования.

1.1. Сопоставление с образцом в C#

Данная возможность была добавлена в C# 7.0. Ранее в C# уже присутствовали конструкции `is` и `switch`. В упомянутой версии компилятора данные конструкции были расширены таким образом, чтобы проверять выражение на соответствие типу и одновременно извлекать эту информацию. [9]

1.1.1. Виды паттернов

На данный момент в C# 7.0 реализовано 4 вида паттернов:

- типовый паттерн;
- константный паттерн;
- wildcard паттерн;
- var паттерн.

Рассмотрим каждый из видов подробнее.

Типовый паттерн имеет вид `Type identifier`. Такой паттерн выполняет две операции: проверяет, имеет ли выражение заданный тип, и преобразует его к этому типу, если проверка прошла успешно. Этот паттерн создает переменную с заданным именем и типом. Переменная получает значение только в том случае, если результат сопоставления с образцом вернул `true`. Проверка осуществляется на этапе выполнения со следующей семантикой: если тестируемое выражение имеет тип, указанный в паттерне, либо оно имеет тип, являющийся наследником заданного, то операция `is` возвращает `true`. Некоторые комбинации статических типов слева от операции и заданного типа являются несовместимыми. В таком случае возникает ошибка времени компиляции. Значение со статическим типом E является паттерн-совместимым с типом T, если типы совпадают, существует неявное преобразование типов, преобразование упаковки, явное преобразование или преобразование распаковки от типа E к типу T. Если ни одного из возможных преобразований нет, то возникает ошибка компиляции.

В качестве константного паттерна может выступать любое выражение константного типа: литералы, имена, объявленные как `const`, константы перечислимых типов, либо выражения `typeof`. Если тестируемое выражение `e` и константа `c` имеют числовой тип, то проверка осуществляется выражением `e == c`, в противном случае используется метод `object.Equals(e, c)`. Таким образом для числовых типов получается избежать преобразований упаковки/распаковки.

Var паттерн выглядит следующим образом: **`var`** `identifier`. Этот паттерн всегда проходит успешно. Новая переменная получает такой же тип, как и статический тип выражения.

Wildcard паттерн — токен _. Он так же всегда проходит успешно и используется в тех случаях, значение переменной не является важным и для него не нужно создавать переменную.

1.1.2. Операция is

Операция `is` имеет форму `relational_expression` **`is`** `pattern`. Она проверяет, соответствует ли `relational_expression` паттерну справа от `is` и возвращает, соответственно, `true` или `false`. Каждый идентификатор из паттерна, создающий новую переменную, получает значение только после того, как `is` возвращает `true`.

Правая сторона операции `is` может быть представлена одним из паттернов описанных в предыдущем подразделе. Пример использования новых возможностей `is` показан в листинге 1.1.1.

Листинг 1.1.1. Использование новых возможностей `is` в C#

```
string ObjectInfo(object o)
{
    if (o is string s)
        return "string: " + s;
    else if (o is int i)
        return "integer: " + i.ToString();
    else if (o is double d)
        return "double: " + d.ToString();
    throw new ArgumentException("Unknown object", nameof(o));
}
```

В данном примере если входной объект является строкой, целым или числом с плавающей точкой двойной точности, то будет возвращена строка с описанием. Иначе возникнет исключение.

1.1.3. Оператор switch

Оператор `switch` уже присутствовал в C# до седьмой версии [10]. Он являлся оператором выбора: в зависимости от значения переданной переменной. Переменные могли быть числового, строкового, символьного, логического или перечислимого типа. В C# 7.0 появилась возможность указывать паттерны вместо значений. Кроме

паттерна появилась возможность использовать условие, при котором ветка оператора считается успешной. Условие указывается после паттерна и обозначается ключевым словом when. Пример использования паттернов в операторе switch приведен в листинге 1.1.2

Листинг 1.1.2. Пример использования паттернов в switch

```
public double GetArea(Shape shape)
{
    switch(shape)
    {
        case Ellipse e when e.Height == e.Width:
            return e.Height * e.Height * Math.PI;
        case Square s: return s.Side * s.Side;
        case Rectangle r: return r.Width * r.Height;
        default: throw new Exception("Unsupported shape");
    }
}
```

1.1.4. Генерация кода для паттернов

Сопоставление с образцом в C# реализовано при помощи замены конструкций, содержащих паттерны, на конструкции уже существующие на данный момент в языке. Рассмотрим код, содержащий паттерны (листинг 1.1.3) и соответствующий ему код, сгенерированный компилятором (листинг 1.1.4).

Листинг 1.1.3. Различные паттерны C#

```
object o = 1;
var b1 = o is int i;
var b2 = o is 1;
var b3 = o is var v;
Console.Write(v);
if (o is string s)
    Console.Write(s);
```

В приведенном в листинге 1.1.3 приведены использования трех видов паттернов: типовый, константный и var паттерны. Первый вид паттернов разворачивается в несколько действий. В первую очередь это объявление вспомогательной переменной `obj` типа `object`. Затем производится присваивание этой переменной и одновременная проверка соответствия типу `int`. Если проверка успешна, создается переменная `i` и ей присваивается значение выражения из паттерна. Константные паттерны переводятся в вызов `object.Equals(value, expression)`. `Var` паттерн переводится в обычное объявление переменной. При использовании типового паттерна внутри условия оператора `if` вместо проверки типа при помощи операции `is`, происходит проверка преобразования типа на `null`. Объявление переменной из паттерна выносится в блок перед оператором `if`.

Листинг 1.1.4. Сгенерированный для паттернов код

```
object o = 1;
// Type pattern
object obj;
if ((obj = o) is int)
    int i = (int)obj;
// Const pattern
bool b2 = object.Equals(1, o);
// Var pattern
object v = o;
Console.Write(v);
// Type pattern in if
string s;
bool flag = (s = (o as string)) != null;
if (flag)
    Console.Write(s);
```

1.1.5. Область видимости переменных в паттернах

Сопоставление с образцом реализовано таким образом, что переменные из паттернов остаются видимыми за пределами конструкций, в которых были использованы сами паттерны. Например, переменная из паттерна, который был использован в условии оператора `if`, становится видна не только внутри тела условного оператора, но и после него. Однако, эта переменная имеет значение только внутри тела `then`, а в теле `else` и после оператора `if` она является неинициализированной. При попытке использования переменной паттерна вне тела `then` будет выдана ошибка компиляции. Однако, этой переменной можно присвоить значение вне паттерна и использовать как обычную локальную переменную.

Для переменных, введенных паттернами в условиях `case` оператора `switch` область видимости ограничена оператором `switch`. Таким образом, для различных случаев `case` нельзя использовать одни и те же имена. Однако, инициализированной переменная считается только внутри действия, соответствующего случаю `case` в котором она была объявлена.

1.2. Сопоставление с образцом в F#

F# является современным функциональным языком программирования. Для функциональных языков сопоставление с образцом является более распространенной и часто используемой возможностью. Кроме того, F# принадлежит к группе языков на платформе .NET, и многие возможности сопоставления с образцом перешли в C# в седьмой версии компилятора.

Язык F# позволяет использовать сопоставление с образцом внутри многих конструкций, например, внутри выражения `match`, связываниях `let`, лямбда-выражениях и обработчиках исключений внутри выражений `try...catch` [11]. Рассмотрим выражение `match`.

```
match expression with
| pattern [ when condition ] -> result-expression
...
```

Каждый паттерн работает как правило преобразования входных данных каким-то образом. В выражении `match` каждый паттерн рассматривается с целью узнать, совместимы ли с ним входные данные. Если совпадение найдено, то выполняется результат выражения. Если совпадения нет, то рассматривается следующий паттерн.

В F# присутствуют различные виды паттернов, они представлены в таблице 1.1

Константный паттерн может быть рассмотрен как аналог оператора выбора из других языков программирования. Входные данные сравниваются со значением паттерна на равенство. Тип литерала должен быть совместимым с типом входных данных.

В случаях когда паттерн является идентификатором, он может быть рассмотрен по-разному. Это может быть значение, помоченное атрибутом `Literal`, конструктор типа-суммы или активный паттерн. Типичным примером может служить извлечение значения из `Optional`, листинг 1.2.1.

Листинг 1.2.1. Пример вывода Option

```
let writeOption (data : int option) =
    match data with
    | Some val  -> printfn "%d" val
    | None -> ()
```

В данном случае, тип `option` определен как тип-сумма с двумя конструкторами: `Some` и `None`. Этот тип используется в случаях, когда необходимо создать значение и, к примеру, вернуть его как результат функции, либо сообщить, что значения нет.

Паттерн `Cons` используется для разбиения списка на первый элемент и остаток списка. В примере (листинг 1.2.2) при помощи

Таблица 1.1. — Виды паттернов в F#

Имя	Описание	Пример
Константный паттерн	Любое числовой, символьный или строковый литерал, константа перечисления	`5.0, “text“, 33`
Идентификатор	Конструктор типа-суммы, метка исключения или активный паттерн	`Some(x), Failure(msg)`
`as` паттерн	pattern **as** identifier	(t1, t2) as tuple
Cons паттерн	Список в функциональном стиле	`head:tail`
Паттерн списка	Сопоставляет со списками `[pattern1;... ; patternN]`	`[a; b; c]`
Паттерн массива	Сопоставляет с массивами	`[| a; b; c; |]`
Паттерн кортежа	Сопоставляет с кортежами `(pattern1,... , patternN)`	`(t1, t2)`
Паттерн wildcard	Сопоставляется любому входу	`(_, t2)`

паттерна извлекается голова списка h и остальные элементы t. Далее происходит печать головы и рекурсивное применение функции к остатку.

Листинг 1.2.2. Пример вывода списка

```
let lst = [ 1; 2; 3; 4 ]

let rec printLst l =
    match l with
    | h :: t -> printf "%d " h; printLst t
    | [] -> printfn ""

printLst l
```

Паттерны списков и массивов предназначены для извлечения их элементов, стоящих на определенных местах. Эти паттерны сопоставляются только спискам или массивам фиксированной длины. Однако, это может быть полезно для некоторых задач, например, вывод вычисление длины вектора (листинг 1.2.3).

Листинг 1.2.3. Пример вычисления длины вектора

```
let vectorLength v =
    match v with
    | [| v1 |] -> v1
    | [| v1; v2 |] -> sqrt (v1*v1 + v2*v2)
    | [| v1; v2; v3 |] -> sqrt (v1*v1 + v2*v2 + v3*v3)
    | _ -> failwith "Unsupported vector length"
```

Кортежные паттерны облегчают работу с отдельными элементами кортежа. Данный паттерн подходит для тех случаев, когда на вход поступают данные, объединенные кортежем, но имеется необходимость произвести какое-то действие с конкретным элементом или несколькими элементами кортежа, при этом игнорируя остальные. Пример использования этого паттерна приведен в листинге 1.2.4.

Листинг 1.2.4. Пример вычисления длины вектора

```
let findZero t =
    match t with
    | (0, 0) -> printfn "All zero"
    | (0, _) -> printfn "First zero"
    | (_, 0) -> printfn "Second zero"
    | _ -> printfn "No zero elements"
```

Паттерн wildcard используется игнорирования переменной паттерна. Полезен для того, чтобы не вводить лишний идентификатор для переменных, которые заведомо не будут использоваться. Кроме того, wildcard паттерн может быть использован не как часть другого паттерна, а как паттерн целиком в конце `match`. Таким образом он послужит условием для действия по умолчанию. В примерах выше показаны оба варианта использования данного паттерна.

ГЛАВА 2

РЕАЛИЗАЦИЯ СОПОСТАВЛЕНИЯ С ОБРАЗЦОМ В СИСТЕМЕ PASCALABC.NET

2.1. Описание подхода к реализации

Синтаксический сахар — это конструкция в языке программирования, которая создается с целью упростить использования каких-либо возможностей языка, а также повысить читаемость программ. Примером синтаксического сахара может служить конструкция `a += b`, эквивалентная записи `a = a + b`. Реализация подобных конструкций нередко упрощается благодаря тому, что синтаксический сахар можно привести к уже имеющимся в языке возможностям, тем самым сводя количество дополнительных семантических проверок к минимуму, либо вовсе избавляясь от них. Однако, для такого подхода необходимо в первую очередь понять, к каким существующим конструкциям языка можно свести новую. Реализуемая конструкция сопоставления с образцом имеет в своей основе две функции:

1. Проверка выражения на соответствие заданному типу, либо возможность привидения к нему.
2. Создание переменной заданного типа и инициализация её соответствующим значением.

Рассмотрим первую функцию. В языке PascalABC.NET уже имеется одноименное выражение `is`, выполняющее эту задачу. Выполнения второй функции можно добиться при помощи оператора **`var`** `<variable name> := <expression>`. Данная конструкция позволяет не указывать тип вводимой переменной, делегируя эту задачу компилятору.

Таким образом, возможность сопоставления с образцом может быть реализована как синтаксический сахар. Следующим этапом является более точное описание кода, в который будут переводиться конструкции сопоставления с образцом.

2.2. Этапы компиляции программы на языке PascalABC.NET

Процесс компиляции можно разделить на несколько крупных этапов, показанных на диаграмме 2.1. На первом этапе работает парсер. На вход ему подается исходный код, на выходе составляется синтаксическое дерево программы. В компиляторе PascalABC.NET используется генератор парсеров GPPG [12]. Данная система генерирует парсеры, разбирающие входные данные снизу-вверх и распознающие языки LALR(1) [13]. Правила для генерации выглядят как показано в листинге 2.2.1. Нетерминал может быть составлен по одному из правил, стоящих справа от двоеточия и разделенных вертикальной чертой. В фигурных скобках указывается соответствующее действие, в случае парсера PascalABC.NET — создание синтаксического узла. В итоге, в конце парсинга будет создано полное синтаксическое дерево программы.

Листинг 2.2.1. Использование типового паттерна в `if`

```
nonterminal :
    sentential form1 { action1 }
  | sentential form2 { action2 }
  ...
  | sentential formN { actionN }
  ;
```

На следующем этапе работа происходит с построенным синтаксическим деревом. Работа с синтаксическим деревом до этапа перевода в семантическое является удобным подходом для реализации различных возможностей языка, оптимизаций, раннего поиска простых семантических ошибок или накопления «легковесной» семантической информации. Синтаксическое дерево имеет меньше информации, чем семантическое, однако, его легче генерировать [14]. Поэтому на данной стадии используются синтаксические конвертеры. Они реализуются с использованием специальных классов — визиторов. Визитор — это паттерн проектирования, реализующий двойную диспетчеризацию и поэтому является стандартным решением для обхода и произведением различных действий над древовидными структурами [15].

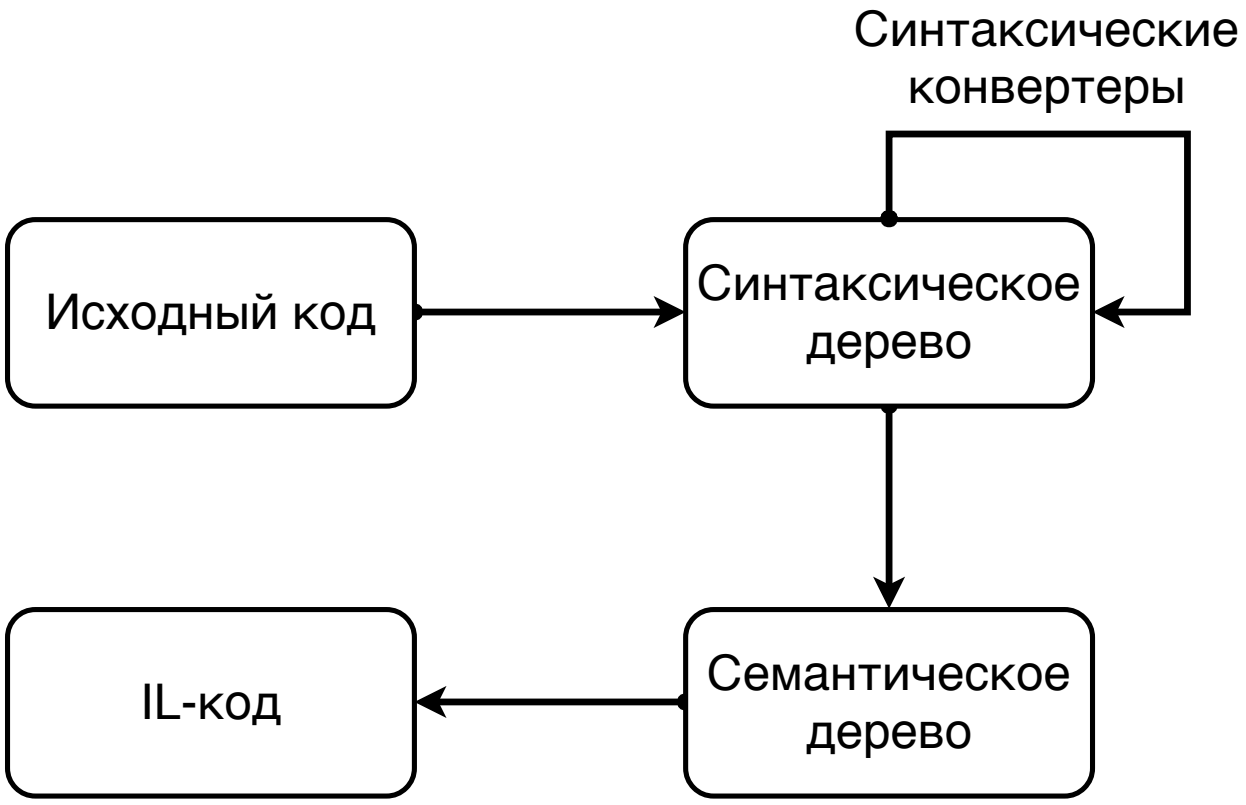

Рисунок 2.1 — Этапы компиляции

Для изменения или обхода синтаксических деревьев в компиляторе PascalABC.NET используются специальные визиторы:

- WalkingVisitor — обход деревьев, возможность совершать дополнительные действия на входе и выходе из узла.
- CollectUpperNodesVisitor — сбор информации о текущем положении визитора в дереве при обходе.
- BaseChangeVisitor — содержит методы для добавления, замены и удаления узлов дерева во время обхода.

2.3. Реализация типового паттерна

2.3.1. Синтаксический вид конструкций сопоставления с образцом

В соответствии с поставленной задачей, паттерны должны поддерживать как проверку выражения на соответствие заданному типу, так и деконструировать выражение на составляющие. Рассмотрим синтаксический вид паттерна, приведенный ниже с использованием расширенных форм Бэкуса-Наура:

```
<type_pattern> ::= <type>(var identifier)
```

В результате при сопоставлении с таким паттерном, будет объявлена переменная с названием `identifier`, имеющая тип <type>. В данном случае типом может быть любой числовой, строковый или символьный тип, а также классы, включая шаблонные.

Конструкция, позволяющая сопоставить выражение с заданным образцом. Возвращает `true` или `false` в зависимости от успеха операции:

```
<expression> is <pattern>
```

Она может быть использована, например, в условии оператора `if` для более короткой проверки типа и дальнейшего использования объекта, с привидением к типу. Пример такого кода приведен в листинге 2.3.1. Предполагается, что класс `Person` определен выше.

Листинг 2.3.1. Использование типового паттерна в `if`

```
// Without type pattern
var o: object;
...
if o is Person then
begin
  var p := o as Person;
  ...
end;

// Using type pattern
var o: object;
...
if o is Person(var p) then
begin
        ...
end;
```

Таким образом, переменная `p` будет создана внутри тела `then`, будет иметь тип `Person` и может быть использована далее в коде.

Оператор выбора действия с использованием паттернов:

```
match <expresion> with
  <pattern1>[when <expression1>]: <statement1>;
  ...
  <patternN>[when <expressionN>]: <statementN>;
  [else defaultStatement]
end
```

Данный оператор схож с оператором выбора действия `case..of`. Выражение <expression> сопоставляется с каждым из паттернов по порядку. Срабатывает только одно действие, первое удовлетворяющее одному из паттернов. Если паттерн сопровождается секцией `when` с условием, то необходимо кроме успешного сопоставления еще и выполнение этого условия. Действие `else` является опциональным и срабатывает только если ни один из предыдущих паттернов не был успешно сопоставлен с выражением.

2.3.2. Сведение новой конструкции к уже имеющимся

Целью типового паттерна является проверка выражения на соответствие заданному типу и создание переменной, имеющей этот тип. Для этой цели была создана функция `IsTest<T>`. Она объединяет в себе задачи проверки типа и присваивание. Код функции приведен в листинге 2.3.2

Листинг 2.3.2. Реализация функции `IsTest`

```
function IsTest<T>(obj: object; var res: T): boolean;
begin
  if obj is T then
  begin
    res := T(obj);
    Result := true;
  end
  else
  begin
    res := default(T);
    Result := false;
  end;
end;
```

Рассмотрим сведение конструкции `is` в условии оператора `if`. В данном случае происходят три действия:

1. условие заменяется на вызов функции `IsTest`;
2. перед условным оператором создается объявление вспомогательной переменной;
3. внутри условия переменной из паттерна присваивается значение вспомогательной переменной.

Код перевода приведен в листинге 2.3.3

Листинг 2.3.3. Использование типового паттерна в `if`

```
if e is Person(var p) then
begin
  ...
end;

// Translates into
var <>genVar: Person;
if IsTest(e, <>genVar) then
begin
  var p := <>genVar;
  ...
end;
```

В свою очередь `match..with` может быть переведен в сахарную конструкцию более низкого уровня: цепочку условных операторов с операцией сопоставления `is` в условиях, как показано в листинге 2.3.4. В квадратных скобках указаны опциональные конструкции.

Листинг 2.3.4. Перевод конструкции `match..with`

```
// Source code
match e with
  Type1(var v1) [when condition1]: statement1;
  ...
  TypeN(var vN) [when conditionN]: statementN;
  else defaultStatement
end;

// Desugared code
if e is Type1(var v1) [and condition1] then
  statement1
else
...
else
if e is TypeN(var vN) [and conditionN] then
  statementN
else
  defaultStatement;
```

2.3.3. Область видимости переменных паттерна

Область видимости переменной, объявленной в паттерне зависит от того, в какой части синтаксического дерева она будет размещена при переводе сахарной конструкции. В самом простом случае переменная может быть помещена в родительский список объявлений переменных. Однако, в таком случае могут возникнуть нежелательные коллизии имен. Например, внутри одного блока нельзя было бы использовать два паттерна, создающие переменные с одним и тем же именем. Поэтому переменные, объявленные в паттернах необходимо помещать как можно ближе к их месту использования.

Область видимости переменных в паттернах ограничивается следующим образом:

- если паттерн используется в условии оператора `if`, то переменные видны в теле `then`;
- если паттерн используется как инициализирующее значение переменной в блоке, то переменные видны внутри этого блока, после объявления инициализируемой переменной;
- если паттерн используется в правой стороне оператора присваивания, то переменные видны после этого оператора внутри блока, которому принадлежит присваивание;
- если паттерн используется операторе `match..with`, то переменные видны в условии `when` и действии, которое соответствует паттерну.

В других контекстах паттерн не появляется из-за ограничений на использование конструкции сопоставления `is`.

2.3.4. Реализация на синтаксическом уровне

Для преобразования сахарной конструкции на синтаксическом уровне был создан специальный визитор `PatternsDesugaringVisitor`. Реализация требует наличия возможностей заменять узлы синтаксического дерева, поэтому визитор был унаследован от `BaseChangeVisitor`. Синтаксические узлы, представляющие операция сопоставления `is` и оператор `match..with` при обходе заменяются на сгенерированные «на лету» поддеревья.

Листинг 2.3.5. Фрагмент реализации визитора

```
public override void visit(is_pattern_expr isPatternExpr)
{
    if (isPatternExpr.right is type_pattern)
        DesugarTypePattern(isPatternExpr);
}

private void DesugarTypePattern(is_pattern_expr isPatternExpr)
{
    // Replace is_pattern with PABCSystem.IsTest function call
    expression expression = isPatternExpr.left;
    type_pattern pattern = (type_pattern) isPatternExpr.right;
    var isTestFunc = SubtreeCreator.CreateSystemFunctionCall("
       IsTest", expression, pattern.identifier);
    ReplaceUsingParent(isPatternExpr, isTestFunc);

    // Declare variable in enclosing statement_list
    AddToAscendantStatementList(new var_statement(pattern.
       identifier, pattern.type));
}
```

2.4. Реализация паттерна деконструкции объекта на составляющие

Следующим этапом является реализация двух возможностей: паттерн деконструирования объекта на составляющие и возможность описывать для классов свою функцию деконструкции. Рассмотрим пример использования возможности деконструкции объекта, вместо типового паттерна.

Листинг 2.4.1. Пример использования паттерна деконструкции

```
// type pattern
if e is Person(var p) then
  Print(p.Name, p.Age, p.Parent);

// deconstructor pattern
if e is Person(var name: string, var age: integer) then
  Println(name, age);
```

Для реализации этого функционала были использованы вспомогательные процедуры с определенным именем. Они могут быть объявлены пользователем, для того чтобы иметь возможность задействовать функционал паттернов деконструкции.

2.4.1. Метод деконструкции объекта

Для деконструирования объекта на его составляющие используется процедура с именем `Deconstruct`. Она должна иметь не менее одного выходного параметра (параметра, помеченного `var`). Для объявленных пользователем классов такая процедура описывается внутри класса. При использовании паттерна деконструкции данная процедура вызывается, чтобы получить значения необходимых переменных. Для примера из листинга 2.5.2 необходимо наличие процедуры `Deconstruct` в классе `Person`, реализованной как показано в листинге 2.4.2. Предполагается, что класс `Person` имеет поля `name`, `age` и `parent`.

Листинг 2.4.2. Реализации процедуры Deconstruct

```
procedure Deconstruct(var name: string; var age: integer; var
   parent: Person);
begin
  name := self.name;
  age := self.age;
  parent := self.parent;
end;
```

Такой тип деконструктора подходит для классов, объявленных пользователем. Однако, необходимо также обеспечить поддержку деконструкции классов, объявленных в стандартной библиотеке .NET, либо из других подключенных библиотек. Деконструкторы для таких объектов могут быть расположены в специальных классах. Таким образом, компилятор должен будет искать методы Deconstruct в объявлениях пользовательских классов и в специальных классах-расширениях. Пример деконструктора для списка — типа, объявленного в библиотеке .NET — описан в листинге 2.4.3.

Листинг 2.4.3. Deconstruct для внешнего типа

```
type ListExt = class
  class procedure Deconstruct<T>(List<T> l; var first: T);
  begin
    if not (l = nil) and l.Count > 0 then
      first := l.First();
  end;
end;
```

2.4.2. Изменения синтаксиса паттерна

Определение паттерна было изменено следующим образом, чтобы позволить указывать несколько переменных, в которые деконструируется объект:

```
<pattern> ::= <type>(<parameters>)
<parameters> ::=
  <parameter> |
  <parameters>,<parameter>
<parameter> ::=
  <variable [:type]> |
  <pattern>
```

Такой вид паттерна позволяет указывать параметры, которые являются именами вводимых паттерном переменных. Тип переменных является необходимо указать в том случае, когда невозможно однозначно вывести тип переменных невозможно. Кроме того, изменения позволяют указывать другой паттерн вместо переменной.

2.4.3. Изменения генерируемого синтаксического дерева

С добавлением паттерна деконструкции и рекурсивных паттернов возникла необходимость внести изменения в `PatternsDesugaringVisitor`. На данном этапе рассматривается три варианта генерации синтаксических поддеревьев для паттернов:

1. вызов экземплярного Deconstruct;
2. вызов специального Deconstruct для внешнего класса;
3. проверка типа (типовый паттерн).

Код для этих вариантов приведен в листинге 2.4.4.

Если тип не указывается пользователем, то его вывод становится задачей компилятора. Места, в которых тип не известен, откладываются до этапа семантического анализа программы. На этом этапе компилятором уже собрана информация о типе, который был указан в паттерне, а также деконструкторы, принадлежащие этому типу. Однако, у такого подхода есть недостатки. При генерации синтаксического дерева, соответствующего коду в листинге 2.4.4, возника-

Листинг 2.4.4. Варианты генерации кода для паттерна

```
// Source
match e with
    Person(var value): <ACTION>
end;

// Desugared code
var <>genVar: Person;
if IsTest(e, <>genVar) then
begin
  // one of 3 cases

  // ********************************************************

  // 1 - instance deconstruct
  var value: {UNKNOWN TYPE}; // should be filled during semantic
     analysis
  <>genVar.Deconstruct(value);

  // 2 - extension deconstruct
  var value: {UNKNOWN TYPE}; // should be filled during semantic
     analysis
  PersonExt.Deconstruct(<>genVar, value);

  // 3 - typecast - when there's only one deconstruction
     parameter and no Deconstruct methods were found
  var value := <>genVar;

  // ********************************************************

  <ACTION>
end;
```

ет необходимость в дополнительных семантических проверках. Более того, на этапе перевода синтаксического дерева в другое синтаксическое информация о типах выражений ограничена. К примеру, невозможно узнать какой из трех вариантов выбрать при переводе паттерна в уже имеющихся конструкции. Поэтому до перевода синтаксического дерева в семантическое откладывается генерация синтаксического поддерева, соответствующего одному из трех случаев, приведенных в листинге 2.4.4.

Перевод синтаксического дерева в семантическое осуществляет визитор `syntax_tree_visitor`. Данный визитор, аналогично другим рассмотренным, обходит дерево от корня к листу. К моменту, когда `syntax_tree_visitor` доходит до использования паттерна, и появляется необходимость выбрать один из трех случаев, уже будет построена часть семантического дерева и собрана некоторая информация, соответствующая текущему синтаксическому дереву. После того, как станет известно, какой случай соответствует встреченному паттерну будет сгенерировано необходимое синтаксическое поддерево. Далее данное поддерево заменит собой синтаксический узел сопоставления с образцом, что приведет к изменению синтаксического текущего дерева. Изменение синтаксического дерева в процессе его обхода визитором `syntax_tree_visitor` может привести к некорректной работе алгоритмов, основывающихся на информации, полученной из исходного синтаксического дерева. Таким образом, при реализации синтаксически сахарных конструкций необходимо создать дерево, которое уже не будет значительно меняться при переводе его в семантическое, если это возможно. Этого можно добиться, если всю необходимую информацию для перевода сахарной конструкции можно собрать перед обходом дерева визитором `syntax_tree_visitor`. Такой подход обеспечивает наибольшую совместимость с другими конструкциями языка, а также избавляет от дополнительных семантических проверок.

Для реализации описанного подхода необходимо свести все три случая к генерации одного синтаксического поддерева. В первых двух случаях для присваивания значений переменным из паттерна используется вызов процедуры `Deconstruct`. Эти случаи можно объединить в один, если объявлять `Deconstruct` для классов из библиотек как методы расширения. Такие методы имеют особый синтаксис: они должны иметь параметр self, который определяет, к какому типу относится этот метод. Основным их преимуществом является то, что их можно вызывать как обычные экземпляр-

ные методы. Таким образом, второй случай сводится к первому. Код `PersonExt.Deconstruct(<>genVar, value)` переводится в такой же, как и в первом случае: `<>genVar.Deconstruct(value)`. Третий случай содержит только присваивание переменной. Для того, чтобы свести этот случай к такому же коду был использован метод расширения для обобщенного типа:

```
procedure Deconstruct<T>(self: T; var res: T); extensionmethod;
begin
  res := self;
end;
```

Семантика методов расширений наследует перегрузку обычных процедур и функций. Методы `Deconstruct` с более конкретными параметрами имеют больший приоритет, чем приведенная выше. Поэтому эта процедура вызовется в том случае, когда других методов `Deconstruct` нет. Такое поведение совпадает с требуемым поведением паттернов. Таким образом типовые паттерны и паттерны деконструкции объекта переводятся в один и тот же код. Обновленный перевод приведен в листинге 2.4.5. Квадратные скобки обозначают возможность указания типа. Если тип не указан, то его вывод откладывается до этапа семантического анализа.

Листинг 2.4.5. Обновленный перевод паттернов

```
// Source
if e is T(var v1[: T1], ..., var vN[: TN]) then
  ...
end;

// Desugared code
var <>genVar: T;
if IsTest(e, <>genVar) then
begin
  var v1[: T1];
  ...
  var vN[: TN];
  <>genVar.Deconstruct(v1, ..., vN);
  ...
end;
```

2.4.4. Реализация на семантическом уровне

После сведения трех случаев паттернов к одному коду, `syntax_tree_visitor` на этапе создания семантического дерева решает только задачу вывода типов, которые не были указаны пользователем. На данном этапе для одного типа может быть объявлено несколько методов `Deconstruct`. Однозначно необходимый метод может быть определен по количеству параметров, либо по их типам, если несколько методов имеют одинаковое количество параметров. Кроме того, необходимо проверить наличие подходящего метода `Deconstruct`. С учетом того, что на этапе перевода паттернов в уже имеющиеся в языке конструкции был сформирован корректный код, после вывода типов можно оставить проверку наличия метода уже реализованным средствам компилятора. Однако, такая ошибка будет выдана в терминах уже нового, переведенного кода. Таким образом, ошибки будут выдавать пользователю особенности реализации воз-

можности, а также быть несвязанными с контекстом, в котором они возникают. Поэтому ошибки, потенциально возникающие в сгенерированном коде, должны быть проанализированы на раннем этапе, чтобы иметь возможность выдать сообщение об ошибке в контексте исходного кода, а не сгенерированного в процессе компиляции.

На этапа семантического анализа кода, сгенерированного при переводе паттернов, компилятору уже известна информация о типе, указанном в паттерне, а также его методах `Deconstruct`. Поэтому в этот момент встраивается алгоритм анализа методов `Deconstruct` и проверяет код на наличие ошибок. Методы `Deconstruct` могут быть обобщенными, как было показано в листинге 2.4.3, но только если это метод расширения. Такое ограничение позволяет вывести типы выходным параметров `Deconstruct` по параметру `self`. Экземплярные методы `Deconstruct` не могут быть обобщенными. Однако, они могут быть объявлены внутри обобщенных классов, так как к моменту вывода типов переменных паттерна обобщенный класс будет инстанцирован. Предположим, что алгоритм анализирует паттерн `e is T(var v1[: T1], ..., var vN[: TN])`. Он совершает следующие шаги:

1. Получение списка всех методов `Deconstruct`, для типа T. Для каждого метода выполняются следующие действия:

    a) если количество параметров метода не равно N, метод считается не подходящим;

    b) если метод является обобщенным, но не является расширением, то возникает ошибка компиляции. Если метод является обобщенным расширением, то типы-параметры выводятся на основе известного типа переменной `self`, она будет иметь тип T;

    c) каждый выходной параметр сравнивается на полное соответствие с типом, указанным пользователем;

d) если все условия выполнены, метод добавляется в список кандидатов.

2. Если список кандидатов пуст, возникает ошибка компиляции.

3. Если кандидатов два или более, то удаляется вспомогательный `Deconstruct`, предназначенный для третьего случая из листинга 2.4.4.

4. Проверка повторяется, если кандидатов по-прежнему два или более, возникает ошибка компиляции. В противном случае имеется единственный подходящий метод `Deconstruct`. Переменные паттерна `v1, ..., vN` получают типы, соответствующие типам выходных параметров метода `Deconstruct`.

2.5. Реализация рекурсивного паттерна

Расширим пример приведенный в листинге 2.4.1.

Листинг 2.5.1. Пример использования паттерна деконструкции

```
// type pattern
if e is Person(var p) then
  Print(p.Name, p.Age, p.Parent);

// deconstructor pattern
if e is Person(var name: string, var age: integer, var parent:
   Person) then
  Println(name, age, parent);
```

В данном примере при использовании паттерна деконструкции, если выражение имеет тип `Person`, то извлекаются сразу поля Name, Age и Parent. Сама по себе такая конструкция может повысить удобство использования переменных паттерна: вместо того, чтобы каждый раз обращаться к полям `p`, можно использовать их напрямую.

Однако, гораздо большую ценность этот функционал представляет в связке с рекурсивными паттернами. Рекурсивные паттерны позволяют использовать результат предыдущего паттерна как входные данные для следующего. Пример использования рекурсивного паттерна представлен в листинге 2.5.2. В этом примере вместо последнего параметра `parent` используется паттерн деконструкции. В результате внутри условного оператора можно получить доступ непосредственно к имени и возрасту родителя.

Листинг 2.5.2. Пример использования рекурсивного паттерна

```
if e is Person(
  var name,
  var age,
  Person(var parentName, var parentAge, var grandParent)) then
  Println(parentName, parentAge);
```

2.5.1. Изменение алгоритма визитора на синтаксическом уровне

Ранее был описан алгоритм сведения конструкции `match..with` к цепочке проверок с использованием операции сопоставления `is` (листинг 2.3.4). Подобным образом рекурсивные паттерны могут быть развернуты в цепочку типовых паттернов и паттернов деконструкции. Предположим, что алгоритм анализирует паттерн `e is T(`v_1`, ..., `v_N`)`. Он совершает следующие шаги: каждый из аргументов $v_1 \ldots v_N$ обрабатывается следующим образом:

1. если аргумент является объявлением переменной (`var `v_i`[: `T_i`]`), то аргумент остается без изменений;
2. если аргумент является паттерном, то v_i заменяется на сгенерированную переменную **var** <>`genVar`$_i$, а к выражению самого внешнего рекурсивного паттерна добавляется выражение

and <>genVar$_i$ **is** v_i и к выражению v_i рекурсивно применяется данный алгоритм.

Таким образом, мы получаем несколько не рекурсивных операций сопоставления `is`, объединенных логическим «И». Пример перевода рекурсивного паттерна приведен в листинге 2.5.3.

Листинг 2.5.3. Пример перевода рекурсивного паттерна

```
if e is Person(
  var name,
  var age,
  Person(var parentName, var parentAge, var grandParent)) then
  Println(parentName, parentAge);

if (e is Person(var name, var age, var <>genVar)) and
   (<>genVar is Person(var parentName, var parentAge, var
      grandParent)) then
  Println(parentName, parentAge);
```

Для сгенерированных переменных не указывается тип, хотя он известен. Это необходимо для поддержки возможности вызывать методы `Deconstruct` предков класса. При поиске подходящего метода `Deconstruct` рассматриваются только методы, относящиеся к данному классу, методы `Deconstruct` его предков игнорируются. Это позволяет использовать методы `Deconstruct` классов-предков для рекурсивных паттернов. Пример использования такой возможности приведен в листинге 2.5.4.

Листинг 2.5.4. Пример перевода рекурсивного паттерна

```
type
  Expr = class end;
  Cons = auto class(Expr)
    r: real;
    procedure Deconstruct(var r: real);
    begin
      r := Self.r
    end;
  end;
  Add = auto class(Expr)
    left,right: Expr;
    procedure Deconstruct(var l,r: Expr);
    begin
      l := left; r := right;
    end;
  end;

function Eval(e: Expr): real;
begin
  match e with
    Add(Cons(l),Cons(r)): Result := l+r;
  end;
end;
```

В данном случае рекурсивным является паттерн `Add(Cons(l),Cons(r))`. Можно перевести эту конструкцию двумя способами:

1.
```
Add(var <>genVar1, var <>genVar2) and
      <>genVar1 is Cons(var c1) and
      <>genVar2 is Cons(var c2)
```

2.
```
Add(var <>genVar1: Cons, var <>genVar2: Cons) and
      <>genVar1 is Cons(var c1) and
      <>genVar2 is Cons(var c2)
```

В первом случае мы оставляем автовывод типа на поздний этап компиляции. Однако, при наличии двух методов `Deconstruct` с одинаковым количеством параметров, алгоритм выбора подходящего метода потребует конкретизировать параметры для однозначного выбора метода. Во втором случае, когда тип параметров указан, алгоритм будет способен вызвать нужный `Deconstruct`, однако для этого случая в классе `Add` должен быть объявлен `Deconstruct` с двумя выходными параметрами типа `Cons`. Более правильным в данном случае будет рассмотреть $\texttt{<>genVar}_i$ как переменные типа `Expr`, так как затем паттерны $\texttt{<>genVar}_i$ **`is`** $\texttt{Cons(}$**`var`** $\texttt{c}_i\texttt{)}$ проверяют переменные $\texttt{<>genVar}_i$ на соответствие необходимому типу.

Поэтому на методы `Deconstruct` было наложено ограничение: класс может быть деконструирован единственным образом и иметь только один метод `Deconstruct`.

ЗАКЛЮЧЕНИЕ

В данной работе был описан процесс и идеи реализации сопоставления с образцом в системе PascalABC.NET. Были успешно внедрены следующие языковые возможности:

1. паттерн проверки типа выражения;
2. паттерн, позволяющий деконструировать объект на его составляющие;
3. рекурсивный паттерн;
4. возможность описывать способ деконструкции объекта на его составляющие единственным образом;
5. расширена операция `is`, позволяющая теперь проверить выражение на соответствие паттерну;
6. оператор `match..with` позволяющий производить различные действия в зависимости от соответствия выражения одному из паттернов.

Реализованные возможности позволяют повысить читаемость кода, упростить часто используемые идиомы, связанные с проверками типов. Кроме того, рекурсивные паттерны дают возможность сопоставлять не только отдельные выражения, но и древовидные структуры. Переопределяемые методы `Deconstruct` помогают включать свою логику в механизм сопоставления с образцом. Оператор выбора действия по паттерну позволяет коротко описать алгоритм выполнения различных действий в зависимости от структуры входных данных.

СПИСОК ЛИТЕРАТУРЫ

1. *Brooker R.A. MacCallum I.R. M. D.* The compiler-compiler. — 1963.
2. About Swift — The Swift Programming Language (Swift 4.2). — URL: https://docs.swift.org/swift-book/index.html (дата обр. 05.06.2018).
3. Система PascalABC.NET. — URL: http://pascalabc.net/ (дата обр. 05.06.2018).
4. Programming Taskbook. — URL: http://ptaskbook.com/ru/index.php (дата обр. 05.06.2018).
5. dotnet/roslyn: The .NET Compiler Platform ("Roslyn") provides open-source C# and Visual Basic compilers with rich code analysis APIs. — URL: https://github.com/dotnet/roslyn (дата обр. 05.06.2018).
6. Система контроля версий GitHub. — URL: https://github.com/ (дата обр. 05.06.2018).
7. GNU Lesser General Public License v3.0 - GNU Project - Free Software Foundation. — URL: https://www.gnu.org/licenses/lgpl-3.0.en.html (дата обр. 05.06.2018).
8. Visual Studio IDE, редактор кода, Team Services и Mobile Center. — URL: https://www.visualstudio.com/ru/ (дата обр. 05.06.2018).
9. Pattern Matching - C# Guide | Microsoft Docs. — URL: https://docs.microsoft.com/en-us/dotnet/csharp/pattern-matching (дата обр. 29.05.2018).

10. *Джозеф Албахари Б. А.* C# 5.0. Справочник. Полное описание языка. — 2013.
11. Pattern Matching (F#) | Microsoft Docs. — URL: `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/pattern-matching` (дата обр. 05.06.2018).
12. *John Gough W. K.* The GPPG Parser Generator. — 2010.
13. *E. K. D.* On the translation of languages from left to right. — 1965.
14. *Aho A. V.*, *Sethi R.*, *Ullman J. D.* Compilers: Principles, Techniques, and Tools. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1986. — ISBN 0-201-10088-6.
15. Design Patterns: Elements of Reusable Object-oriented Software / E. Gamma [и др.]. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. — ISBN 0-201-63361-2.