Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Факультет математики, механики и компьютерных наук

Направление подготовки 010400 — «Информационные технологии»

# МОДЕЛЬНЫЙ ЯЗЫК ПРОГРАММИРОВАНИЯ С λ-ФУНКЦИЯМИ

Курсовая работа
студентки
Ю. В. Беляковой

Научный руководитель:
доцент, к.ф.-м.н.
С. С. Михалкович

Ростов-на-Дону
2011

# Оглавление

# 1. Введение

В последнее время в императивные языки, например, C#, Visual Basic .NET, C++0x, Python стали включать функциональные элементы, в частности, лямбда-выражения. Под л**ямбда-выражением** понимается специальная синтаксическая конструкция, которая позволяет определять анонимные функции на месте функциональных переменных. Фактически такая конструкция является весьма удобным безымянным аналогом обычных функций, принятых в императивных языках.

В **лямбда-исчислении** функции строятся посредством ***λ-абстракций***: $\lambda x.t$. Данную λ-абстракцию можно интерпретировать как функцию одного переменного $x$, которая возвращает выражение $t$. Функция нескольких переменных строится как функция одного переменного, которая возвращает функцию нескольких переменных «меньшей размерности». Например, функция двух переменных $x$, $y$, вычисляющая произведение $x \cdot y$, может быть записана следующим образом: $\lambda x.\lambda y.x * y$. λ-абстракцию $\lambda x.t$ можно ***применить*** к некоторому аргументу $y$: $\lambda x.t\ y$. Применение означает подстановку выражения $y$ в выражение $t$ вместо переменной $x$. Отсюда вытекает важное отличие λ-абстракций от функций, принятых в императивных языках: λ-абстракции могут применяться к различному числу аргументов, и от этого зависит тип возвращаемого значения применения, в то время как набор аргументов «императивных функций» жестко задан, равно как и тип возвращаемого значения.

Такое *преобразование* функции *n* аргументов, что она может быть вызвана как цепочка функций, принимающих по одному аргументу, называется **каррированием**. Вызов каррированной функции от неполного набора аргументов называется **частичным применением**. Каррирование и частичное применение характерны в первую очередь для функциональных

языков. Однако соответствующие механизмы стали появляться и в императивных языках. Например, так происходит каррирование в C#:

```
int add(int x, int y)
{
     return x + y;
}

Func<int, Func<int, int>> curryAdd = x => (y => add(x, y));
Func<int, int> add3 = curryAdd(3);      // частичное применение
int x = curryAdd(3)(4);       // вызов каррированной функции
int y = add3(4);
```

Переменные `x`, `y` будут содержать число 7. Функции имеют следующие типы:

```
add: (int, int) -> int; curryAdd: int -> int -> int;
add3 (== curryAdd(3)): int -> int;
```

Основной темой данной работы является проверка и вывод типов в императивном языке с элементами функционального. В частности, подразумевается реализация функций в стиле каррированных функций функциональных языков, а также простой механизм для частичного применения.

Ещё одним направлением исследования стали **шаблоны функций**. Сейчас существует два основных подхода к реализации шаблонов — шаблоны C++ и универсальные шаблоны .NET [1].

**Шаблоны C++**. Для параметров шаблона разрешены все синтаксически правильные конструкции. Шаблоны подвергаются проверке синтаксиса, а проверка семантики откладывается на этап инстанцирования и проводится для каждого набора конкретных типов параметров с нуля. Шаблоны фактически хранятся на уровне синтаксиса.

**Универсальные шаблоны .NET**. В .NET принята другая концепция: по умолчанию для параметров шаблонов разрешены только операции, допустимые для типа *Object* — предка всех классов, а для использования специфических операций необходимо явно указывать ограничения, которым должны удовлетворять параметры шаблонов. При таком подходе этап

инстанцирования как таковой отсутствует, для использования некоторого типа в качестве параметра шаблона нужно лишь проверить, что тип удовлетворяет заданным ограничениям.

В работе исследуется альтернативный подход к реализации шаблонов. Предлагается разрешить для параметров шаблона все синтаксически корректные операции, но проводить как можно более подробный семантический анализ тела шаблона (к примеру, обнаруживать повторно объявленные идентификаторы) с целью выявления семантических ошибок на более раннем этапе, нежели инстанцирование, а также повышения эффективности данного этапа.

# 2. Постановка задачи

1) Разработать модельный императивный язык программирования, поддерживающий описание и вызов функций, лямбда-выражения и шаблоны функций.

2) Разработать грамматику языка.

3) Создать front-end и middle-end части компилятора. Основной задачей является проверка и вывод типов, генерация или исполнение кода на данном этапе не производится. Также не поддерживаются лямбда-выражения и вызов шаблонов функций.

4) Разработать алгоритм поиска типа

5) Создать структуру таблицы имен

6) Найти подход к указанной реализации шаблонов функций.

# 3. Модельный язык PollyTL

Язык PollyTL — это простой императивный язык с функциями, шаблонами функций и лямбда-выражениями.

## 3.1 Типы

Система типов состоит из трех базовых типов: `Int`, `Double`, Bool и функциональных типов, получаемых как всевозможные комбинации базовых. Например:

```
(Int -> Int) -> (Int -> Int -> Bool -> Int).
```

Функциональные типы правоассоциативны, т.е. тип `Int -> Int -> Int` эквивалентен типу `Int -> (Int -> Int)`.

Тип Int — это целые числа. Double — числа с плавающей точкой. Bool — логический тип, представлен константами `true` и `false`.

Также неявно присутствует специальный тип `void`. Он нужен для описания функций без параметров и процедур — функций без возвращаемого значения. Этот тип может присутствовать и в типе переменных, но значение типа `void` явно написать нельзя.

**Операции над типами**.

Для выражений типа Bool допустима унарная операция *!* (отрицание) и бинарные операции: || (или), *&&* (и).

Для целых чисел допустимы следующие бинарные операции: +, –, *, */* (деление, результатом является выражение типа Double), ***div*** (деление нацело), ***mod*** (взятие остатка от деления). *Div* и *mod* — ключевые слова.

Для выражений типа Double допустимы операции *+, –, *, /*.

Для выражений одного типа можно использовать сравнения на равенство == и неравенство *!=*. Для числовых выражений одного типа также доступны сравнения <, <=, >, >=. Результат имеет логический тип.

## 3.2 Структура программы

Программа имеет следующий вид:

*<секция описаний>*
***main***
    *<операторы>*
***end***

Секция описаний включает описание переменных, констант, функций и шаблонов функций.

## 3.3 Комментарии

Однострочные комментарии отделяются символами //.

Многострочные комментарии заключаются в /* */. Многострочные комментарии могут быть вложенными.

## 3.4 Описание переменных и констант

Для описания констант используется следующий синтаксис: ***const*** *<тип> <имя переменной> = <выражение>;*

Например:

```
const Int SZ = 100;
```

Для описания переменных предусмотрены два способа.

1) Описание переменных с явным указанием типа

*<тип> <список переменных>;*
*<список переменных> ::= <описание переменной>*
*| <список переменных>, <описание переменной>*
*<описание переменной> ::= <имя переменной> [= <выражение>]*

Например:

```
Int a, b = 5, c = b*b, d;
```

2) Описание переменной с автоматическим выводом типа

***var*** *<имя переменной> = <выражение>;*

Например:

```
var fb = 4 / 3 + 6.7;
```

## 3.5 Определение функций

При определении функции тип возвращаемого значения можно не указывать. В этом случае он будет выведен автоматически. Синтаксис следующий:

*[<тип возвр. значения>]* ***fun*** *(<список формальных пар-ов>)*
*<операторы>*
***end***
*<список формальных пар-ов> ::= <пусто>*
*| <список формальных пар-ов>, <формальный пар-р>*

*<формальный пар-р> ::= <тип> <имя параметра>*

Если функция возвращает значение, то в теле должен присутствовать оператор ***return*** *<выражение>;* Приведем пример функции, вычисляющей квадрат суммы двух чисел:

```
fun iSumSqr(Int a, Int b)
     var s = a + b;
     return s * s;
end
```

Эта функция имеет тип `Int -> (Int -> Int)`. Она может быть присвоена переменной функционального типа:

```
Int -> (Int -> Int) f = iSumSqr;
var f1 = iSumSqr;
```

Процедура определяется аналогично:

```
void fun p()
     // операторы
end
```

## 3.6 Определение шаблонов функций

Определение шаблона функции очень похоже на определение функции, добавляется только указание списка параметров шаблона. Список параметров заключается в специальные символы *[!<список параметров>]* и записывается после имени шаблона функции. Например:

```
fun sqr[!T](T x)
     return x * x;
end
```

## 3.7 Лямбда-выражения

**Лямбда-выражение** — это синтаксическая конструкция, позволяющая определить функцию на лету. Лямбда-выражение может быть передано в функцию в качестве параметра или присвоено функциональной переменной. Синтаксис таков:

*<параметры> => { <тело> }*

Если параметров несколько, их надо заключать в круглые скобки. Типы параметров можно не указывать.

Тело лямбда-выражения может состоять из одного выражения либо из последовательности операторов. В последнем случае тело обязательно должно содержать оператор *return*. Приведем несколько примеров:

```
Int -> Int iSqr = Int x => x*x;
var sqr = x => x*x;
var someLambda = (x, y) => {
     x = 2 * x;
     return x + y;
};
```

## 3.8 Операторы

**Оператор присваивания** имеет следующий синтаксис:

*<переменная> = <выражение>;*

Выражение должно иметь тот же тип, что и переменная. Например:

```
Int a;
a = (87 * 13 + 1032) mod 43;
Int -> Int iSqr;
iSqr = Int x => x*x;
```

**Условный оператор**

***if*** *<условие$_1$>* ***then***
    *<операторы$_1$>*
***{elif*** *<условие$_i$>* ***then***
    *<операторы$_i$>}*
***[else***
    *<операторы$_2$>]*
***fi***

В качестве условия указывается некоторое логическое выражение. Elif- и else-части оператора являются необязательными. Пример:

```
if a < 0 then
     b = -b;
elif (a > 0) && (a < 10) then
     b = b * 2;
else
     b = 0;
fi
```

**Оператор цикла с предусловием**

***while*** *<условие>* ***do***
    *<операторы>*
***endw***

Пример:

```
var i = 0;
while i < 10 do
     i = i + 1;
     // делать что-нибудь
endw
```

**Вызов функций**. Синтаксис следующий:

*<имя функции>(<список фактических параметров>)*
*<список фактический параметров> ::= <список выражений>*

Функции могут быть вызваны с различным числом параметров, не превышающим число формальных параметров в описании. От списка фактических параметров зависит тип возвращаемого значения функции. Приведем несколько примеров:

```
fun iSum(Int a, Int b)
     return a + b;
end
var add2 = iSum(2);       // add2: Int -> Int
var c = add2(3);          // c : Int

// применяет функцию одного аргумента f к x
fun app(Int -> void f, Int x)
     f(x);
end
```

## 3.9 Функция println

На данный момент единственным исполняемым оператором является вызов функции `println`. Эта функция принимает любое число выражений и выводит на консоль их типы. Например, результатом вызова `println(iSum, iSum(2), add2, c)` из предыдущего примера будет:

```
iSum : Int -> (Int -> Int)
iSum(2) : Int -> Int
add2 : Int -> Int
c : Int
```

# 4. Грамматика языка и парсер

Лексический анализ программы проводится с помощью генератора сканеров (лексических анализаторов) *GPLex* и генератора парсеров (синтаксических анализаторов) *GPPG*. **GPPG** (The Gardens Point Parser Generator) [2] — это свободно распространяемый генератор *LALR(1)* парсеров. Результатом работы генератора является парсер на языке C#.

**LALR(1)** (LookAhead Left Recursive) — восходящий алгоритм синтаксического разбора. Распознает подкласс контекстно-свободных языков.

Грамматика языка разработана с учетом всех заявленных в пункте 3 конструкций и приведена в Приложении 1.

В случае несоответствия программы заданной грамматике парсер прерывает анализ при обнаружении первой ошибки.

# 5. Синтактико-семантическое дерево

## *5.1 Назначение*

В процессе синтаксического анализа программы строится синтактико-семантическое дерево. Оно отражает *структуру программы* и может содержать дополнительную информацию, необходимую для семантического анализа.

В момент создания узлов дерева не проводится никаких семантических проверок, но на этом этапе может происходить выявление некорректных синтаксических конструкций, которые пропускаются грамматикой. Так, например, список параметров лямбда-выражения допускает наличие параметров без указания типов, а при описании функции типы формальных параметров обязательны. В грамматике языка это различие не учитывается, поэтому проверку корректности формальных параметров функции надо проводить явно.

## *5.2 Реализация синтаксического дерева*

Поскольку в момент формирования дерева семантика не учитывается, в данном контексте будем называть его *синтаксическим*.

**Узлы** синтаксического дерева соответствуют некоторым синтаксическим конструкциям языка. Они представлены специальными классами, являющимися наследниками класса `SyntaxSemanticTreeNode`.

```
class SyntaxSemanticTreeNode
```

- *CodeSpanLocation* SourceContext

Поле `SourceContext` содержит информацию о размещении соответствующего элемента в тексте программы. Данные о размещении элемента предоставляются парсером во время синтаксического анализа.

Для примера рассмотрим класс, соответствующий конструкции описания функции:

`class` **`ExplicitFunctionDeclaration`**

- *`Identifier`* `Name — имя функции`
- *`TypeRepresentation`* `ResultType — тип возвращаемого значения`
- *`FormalParametersList`* `Parameters — список параметров`
- *`StatementsList`* `Statemnts — тело функции`

В иерархии классов узлов дерева можно выделить четыре основных категории: описания, типы, выражения, операторы. В общей сложности реализовано порядка 60 различных классов.

## 5.3 Проверка семантики

После того как синтаксическое дерево полностью построено, запускается алгоритм его обхода для проверки семантики. Обходом занимается специальный класс `SemanticVisitor` содержащий методы для обработки узлов дерева. Например:

```
public void Visit(CallingExpression callExpr)
{
    ProgramSemantics.CheckIdentExists(callExpr.Name);
    callExpr.SemanticCheckCallable();
    Visit(callExpr.FactParameters);
    callExpr.SemanticCheckParameters();
}
```

**Замечание**: узлы дерева одного типа могут обрабатываться по-разному в зависимости от конструкций, в которых они используются.

В процессе обработки узлов они могут дополняться некоторой семантической информацией, ненужные данные могут сбрасываться. Классы синтактико-семантического дерева содержат методы для семантических действий, которые вызываются при обходе узлов. Такими методами явля-

ются, например, методы `SemanticCheckCallable` и `SemanticCheckParameters` класса `CallingExpression` из листинга выше.

Далее рассмотрим коротко **основные виды семантических проверок**.

1) ***Проверка используемых идентификаторов***. Сюда включается отслеживание повторно объявленных идентификаторов, использование необъявленных идентификаторов.

Пространствами имен являются: основная программа (блок внутри `main` – `end` и секция описаний перед `main`), функция, шаблон функции, разделы операторов внутри условного оператора и тело цикла `while`. Имена внутри пространства имен должны быть уникальными.

2) ***Проверка типов***. При выводе типов выражений необходимо проверять корректность типов подвыражений. К примеру, операцию *div* можно применять только к выражениям целого типа. Вызывать можно функции и функциональные переменные, но никак не логические. При использовании оператора присваивания надо проверять совпадение типов переменной и выражения, а при вызове функций сравнивать типы формальных и фактических параметров.

3) ***Вывод типов***. Вывод типа выражения нужен, в первую очередь, при описании переменной через **`var`**. Также вывод происходит при анализе функций с неопределенным возвращаемым значением.

4) ***Анализ операторов return***. Если функция имеет возвращаемый тип, отличный от `void`, возврат значения должен производиться всегда. Поэтому необходимо анализировать тело функции на наличие «пустых» веток. Например, такая функция должна вызвать ошибку компиляции:

```
Bool fun failEven(Int x)
     if x mod 2 == 0 then
          return true;
     fi
end
```

Правильным решением будет:

```
Bool fun even(Int x)
     if x mod 2 == 0 then
          return true;
     else
          return false;
     fi
end
```

Или:

```
Bool fun even(Int x)
     if x mod 2 == 0 then
          return true;
     fi
     return false;
end
```

Все операторы `return` должны содержать выражения одного типа, совпадающие с заявленным возвращаемым типом.

Если возвращаемый тип функции не определен, он выводится по выражению в операторе return, если в теле функции есть хоть один такой оператор. Иначе возвращаемым типом считается `void`. Как и в первом случае, выражения в операторах возврата значения должны иметь один тип.

**Замечание**: Лямбда-выражения и шаблоны функций в текущей версии компилятора не реализованы на уровне семантики.

# 6. Таблица символов

Как было отмечено, на этапе семантического анализа необходимо следить за использованием в программе идентификаторов. Для этого используется **таблица символов**.

Таблица символов, точнее, таблица идентификаторов, хранит семантическую информацию обо всех используемых в программе идентификаторах — переменных, именах функций. Для каждого пространства имен создается собственная таблица символов. На данный момент в языке существует два типа пространств имен: пространство имен функции и пространство имен операторного блока.

Таблица идентификаторов пространства имен представляет собой хэш-таблицу. Ключами являются строки идентификаторов, а значениями

— объекты специального класса. Таблица идентификаторов реализует интерфейс `ISymbolIdentifierTable`.

```
/// Таблица символов-идентификаторов
public interface ISymbolIdentifierTable
{
     /// Тип контекста
     NameContextType ContextType { get; }

     /// Проверяет наличие в таблице идентификатора identNode
     bool Contains(IIdentifier identNode);

     /// Добавляет в таблицу символов запись о новом
     /// идентификаторе identNode типа identType
     SymbolIdentifierTableRecord Add(
          SymbolIdentifierType identType,
          SyntaxSemanticTreeNode identNode);

     /// Возвращает атрибуты идентификатора в таблице символов
     SymbolIdentifierTableRecord Get(IIdentifier identNode);
}
```

Запись в таблице символов (то есть значение хэш-таблицы) является объектом класса `SymbolIdentifierTableRecord`. В записи содержится информация о *mune* идентификатора и объект класса атрибутов идентификатора.

`class` **`SymbolIdentifierTableRecord`**

- *`SymbolIdentifierType`* `IdentType`
- *`SymbolIdentifierTableAttributes`* `Attributes`

Идентификатор может иметь один из следующих типов: константа, переменная, функция, шаблон функции, формальный параметр. Атрибуты идентификатора содержат информацию о его типе, а также по необходимости другую информацию, характерную для данной категории идентификатора. Классы атрибутов идентификаторов наследуются от класса `SymbolIdentifierTableAttributes`.

В процессе семантического анализа таблицы символов про-странств имен помещаются и удаляются со стека таблиц. Если начинается обход нового пространства имен (функции или операторного блока), создается новая таблица идентификаторов и помещается на вершину стека. Когда

обход соответствующего блока завершен, таблица снимается со стека. Записи, соответствующие функциям, содержатся только в главной таблице идентификаторов, поскольку вложенное описание функций не предусмотрено.

**Алгоритм поиска имени в структуре таблиц символов**. Таблица идентификаторов реализуется с помощью хэш-таблицы, поэтому поиск в ней осуществляется за константное время. Пространства имен являются вложенными, соответствующие таблицы символов хранятся в стеке. Сначала имя ищется в таблице символов на вершине стека. Если имя не найдено, рассматривается пространство имен верхнего уровня. Поэтому в общем случае поиск имени имеет сложность $O(d)$, где $d$ — глубина стека, т.е. уровень вложенности пространств имен текущего места в программе.

# 7. Таблица типов

Система типов языка весьма ограничена. Она состоит из базовых *именованных* типов и всевозможных *функциональных* типов.

**Функциональный тип** — это тип $S \rightarrow U$ («из S в U»), где S и U — это некоторые допустимые типы.

Заметим, что различные функциональные типы могут содержать внутри себя одни и те же типовые компоненты. Рассмотрим такую программу:

```
fun iSqr(Int x)
     return x * x;
end

fun iSum(Int a, Int b)
     return a + b;
end

// сумма "обработанных" чисел
fun appSum(Int -> Int f, Int a, Int b)
     a = f(a);
     b = f(b);
     return iSum(a, b);
end
```

```
main
     Int a = 345, b = 675;
     var sqrSum = iSqr(iSum(a, b));
     var sumSqr = appSum(iSqr, a, b);
     println(iSqr, iSum, appSum);
     var sumSqrF = appSum(iSqr);
     println(sumSqrF);
     var sumSqr1 = sumSqrF(a, b);
end
```

Посмотрим на типы:

```
iSqr : (Int -> Int)
iSum : (Int -> (Int -> Int))
appSum : ((Int -> Int) -> (Int -> (Int -> Int)))
sumSqrF : (Int -> (Int -> Int))
```

Тип `(Int -> Int)` встречается здесь *5* раз, более сложный тип `(Int -> (Int -> Int))` — *3* раза. Возможно, пример несколько надуманный, но он показывает, что части сложных функциональных типов весьма вероятно могут встречаться неоднократно. Обозначим имеющиеся функциональные типы следующим образом:

```
Int -> Int = S
Int -> (Int -> Int) = Int -> S = U
(Int -> Int) -> (Int -> (Int -> Int)) = S -> U = T
```

Таким образом, в нашей программе используется всего лишь 4 типа: Int, S, U и T.

Исходя из приведенных соображений, было решено использовать на этапе семантического анализа ***единую таблицу используемых типов (ТИТ)***. Это динамический массив атрибутов типов. В процессе обхода узлов типов синтактико-семантического дерева в таблицу добавляются новые типы, а информация о типе в этих узлах заменяется на соответствующий индекс таблицы используемых типов. Если такой тип ранее уже использовался, то только сохраняется соответствующий индекс. В атрибутах идентификатора таблицы символов в качестве типа также используется индекс записи в таблице типов.

Таким образом, проверка типов на этапе семантического анализа сводится к сравнению индексов — целых чисел. Но *построение* таблицы типов также должно быть эффективно.

При построении таблицы основной задачей является **поиск типа заданной структуры**. Сначала рассмотрим простейший случай: мы обрабатываем узел дерева *именованного* типа. Введем хэш-таблицу используемых

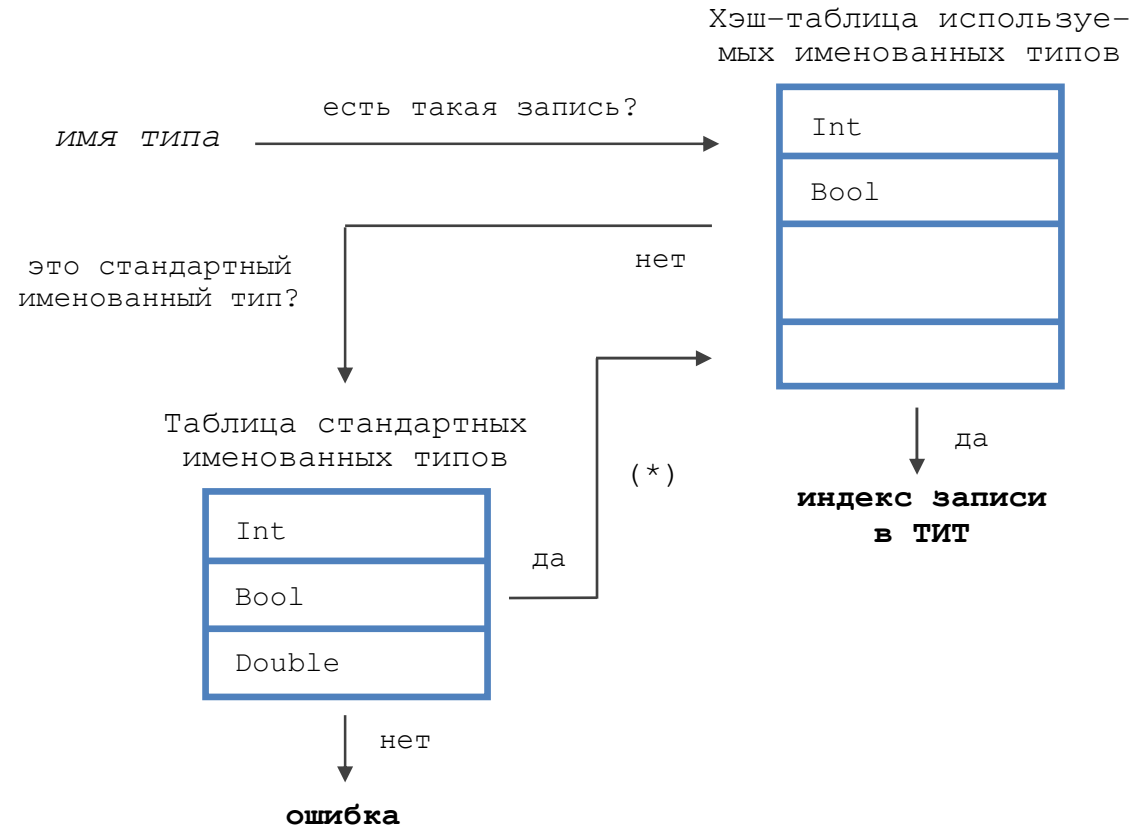


*Рисунок 1. Алгоритм поиска именованного типа*

именованных типов. Ключом является имя типа, а значением — индекс записи типа в таблице используемых типов (динамическом массиве атрибутов типа). Тогда получаем алгоритм, приведенный на Рисунке 1.

За счет использования хэш-таблиц поиск именованного типа требует O(1) операций.

Теперь рассмотрим случай с *функциональным* типом. Он состоит из двух частей: типа аргумента и типа возвращаемого значения. В простей-

шем случае оба типа являются именованными, но могут быть и функциональными. Пусть мы уже знаем индексы этих типов в ТИТ — *a* и *b*. Необходимо добавить тип *a* –> *b* в таблицу используемых типов, если данный тип ранее не использовался, и уметь быстро найти его там.

Воспользуемся следующей структурой: заведем хэш-таблицу, ключами которой являются *индексы типов аргументов* функционального типа, значениями — тоже хэш-таблицы. Найдем хэш-таблицу, соответствующую ключу *a*, если её нет (то есть тип *a* –> *X* ранее не встречался) — добавим новую. Ключами «внутренней» хэш-таблицы будут *индексы типов возвращаемых значений* функционального типа (в нашем случае это *b*), а значениями — индексы всего функционального типа *a* –> *b* в таблице используемых типов. Если «внутренняя» хэш-таблица не содержит записи для

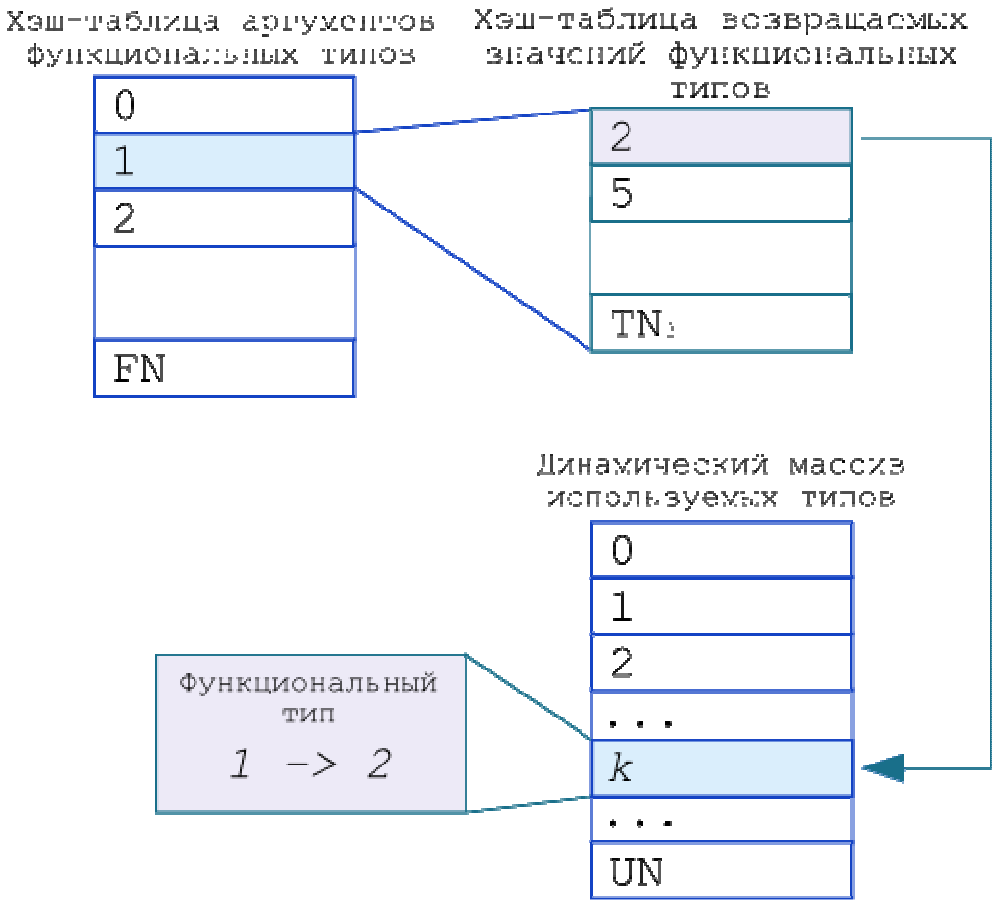


*Рисунок 2. Структура таблиц функциональных типов*

ключа *b*, добавляем тип *a* –> *b* в ТИТ и создаем соответствующую запись в хэш-таблице. Схема предложенной структуры приведена на Рисунке 2.

**Элементарным** типом назовем именованный тип. **Длиной** функционального типа назовем число элементарных типов в его структуре (например, длина типа `Int -> Bool` равна двум, а `(Int -> Int) -> (Bool`

`-> (Int -> Int))` — пять). В синтаксическом дереве узел функционального типа представлен списком типов. Поиск соответствующего типа в ТИТ (или формирование, если данный тип ранее не использовался) требует $O(m)$ операций, где $m$ — длина типа.

Поиск или формирование функционального типа происходит в двух случаях: 1) при обходе узла функционального типа синтаксического дерева (соответствует описанию функциональной переменной с явным указанием типа); 2) при обходе узла функции синтаксического дерева: в этом случае тип формируется по списку формальных параметров и типу возвращаемого значения. Во всех остальных случаях *проверка типов* происходит эффективно и *сводится к сравнению целых чисел*.

Благодаря такой организации таблиц типов, получаем простой алгоритм проверки типов при *частичном применении*.

```
Вход:
FT — тип функции (FT.From — тип аргумента, FT.To — тип возвра-
     щаемого значения);
PTList — список типов фактических параметров;

Алгоритм:
цикл по PTList
     если FT не является функциональным типом:
          ошибка
     // currT — текущий элемент списка
     если  currT != FT.From:
          ошибка
     иначе
          FT := FT.To
возвращаемый_тип_функции := FT.To
```

Сложность алгоритма составляет $O(p)$, где $p$ — число фактических параметров. Синтаксически частичное применение является вызовом функции с меньшим набором аргументов.

# 8. Алгоритмы обработки шаблонов функций

## *8.1 Введение*

Шаблоны C++ являются более гибким механизмом, нежели подход, используемый в универсальных шаблонах .NET. Основной их недостаток заключается в том, что тело шаблона не подвергается семантическому анализу, в результате чего на этап инстанцирования выносится довольно большой объем работы, причем многократной, поскольку проверка семантики выполняется для каждого набора конкретных типов параметров.

Предложим альтернативный подход к реализации шаблонов на примере модельного языка PollyTL:

- в процессе обхода тела шаблона будем проводить частичный семантический анализ, который включает: обнаружение необъявленных идентификаторов; выявление повторно объявленных идентификаторов; частичный анализ оператора return; полную семантическую проверку выражений и операторов, не содержащих подвыражений, типы которых включают параметры шаблона.
- построим исчерпывающий набор ограничений, которым должны удовлетворять параметры шаблона, чтобы инстанция от набора конкретных типов была семантически корректной.
- на этапе инстанцирования будем проверять, удовлетворяют ли указанные параметры построенным ограничениям.

Алгоритмы построения и обработки множества ограничений предложены Бенжамином Пирсом. В [2] он рассматривает их на примере языка, являющегося подмножеством типизированного λ-исчисления. Далее будут рассмотрены модификации данных алгоритмов, которые можно использовать для реализации предложенной концепции шаблонов функций языка PollyTL.

Тип *X*, не являющийся простым типом или функциональным типом, составленным из простых (то есть тип, для которого не существует операций создания значений данного типа), будем называть **типовой переменной**. Типовые переменные можно **конкретизировать** другими типами с помощью **подстановки**.

**Подстановкой типов** называется конечное отображение из типовых переменных в другие типы. Например, подстановка, сопоставляющая тип Int переменной X и T переменной Y записывается так: $\sigma = [X \mapsto Int, Y \mapsto T]$. Запись $dom(\sigma)$ обозначает множество типовых переменных в левой части пар, образующих $\sigma$, а $range(\sigma)$ — множество типов, встречающихся в правой части.

Применение подстановки определяется так:

$$\sigma(X) = \begin{cases} T \text{ если } (X \mapsto T) \in \sigma \\ X \text{ если } X \notin dom(\sigma) \end{cases}$$

$$\sigma(Int) = Int$$

$$\sigma(Bool) = Bool$$

$$\sigma(Double) = Double$$

$$\sigma(T_1 \to T_2) = \sigma(T_1) \to \sigma(T_2)$$

**Композицией подстановок** $\sigma$ и $\gamma$ назовем подстановку, которая получается по следующим правилам:

$$\sigma \circ \gamma = \left[ \begin{array}{l} X \mapsto \sigma(T) \text{ для всех } (X \mapsto T) \in \gamma \\ X \mapsto T \text{ для всех } (X \mapsto T) \in \sigma \text{ при } X \notin dom(\gamma) \end{array} \right]$$

В этих определениях *параметры шаблонов* функций мы будем интерпретировать как *типовые переменные*. Исходя из конструкций, допустимых в теле шаблона, необходимо построить *набор ограничений*, которым должны будут удовлетворять конкретные типы. При вызове шаблона функции от конкретных типов будем проводить проверку этих типов и выполнять их *подстановку* вместо параметров шаблона (типовых переменных).

## 8.2 Построение ограничений

Определим **ограничения на типы**, возможные в нашем языке. Они могут иметь один из двух видов:

1) ***S = T***, где S и T — это типы или типовые переменные.

Такое ограничение может быть получено, если, например, в программе встречен оператор `x = 5.0 * 2`, где *x* имеет тип параметра шаблона *X*. В этом случае получаем ограничение *X = Double*, так как выражение *5.0 * 2* имеет тип Double. Другой пример: `x(t)`, где *x: X*, *t: T*. Тогда ограничение имеет вид *X = T –> Y*, где Y — новая типовая переменная. Этот вид ограничений назовем ***первичным***.

2) ***S ∈ { $T_i$ } i = 1..n — вторичное*** ограничение.

Например, оператор `x = 2 * x`, где *x: X* означает, что *X ∈ { Int, Double }*, так как операция * определена для обоих типов.

Приведем полный набор правил построения ограничений в зависимости от вида конструкции языка. Через *type(e)* будем обозначать тип выражения *e*. Ограничения вида *X = X* при построении будем отбрасывать, так как они не несут никакой значимой информации. Если ограничение имеет вид *S = T*, где S и T это конкретные типы (не содержащие типовых переменных), то это просто проверка типов: если типы совпадают, ограничение отбрасывается, иначе — семантическая ошибка. Через ; будем писать список конструкций, для которых ограничения эквиваленты.

1) `a + b ; a - b ; a * b ; a / b` **=>**
   *type(a) ∈ {Int, Double}, type(b) ∈ {Int, Double}*
2) `a div b ; a mod b` **=>** *type(a) = Int, type(b) = Int*
3) `!a ; a || b; a && b` **=>** *type(a) = Bool, type(b) = Bool*
4) `a == b ; a != b` **=>** *type(a) = type(b)*
5) `a < b ; a <= b; a > b; a >= b` **=>** *type(a) = type(b), type(a) ∈ {Int, Double}*
6) `T x = expr ; const T x = expr` **=>** *type(expr) = T*

7) `x = expr` | `x: X` => $X = type(expr)$

8) **`if`** $expr_1$ **`then`**
…
**`elif`** $expr_2$ **`then`**
…
**`elif`** $expr_n$ **`then`**
…
**`else`**
…
**`fi`** => $type(expr_1) = Bool, type(expr_2) = Bool, \ldots, type(expr_n) = Bool$

9) **`while`** `expr` **`do`**
…
**`endw`** => $type(expr) = Bool$

10) `f()| type(f) не является конкретным типом` =>

$type(f) = void \rightarrow X$ | X — новая типовая переменная

11) `f(p1, p2, …, pn)` | `type(f) не является конкретным типом` =>

$type(f) = type(p1) \rightarrow (type(p2) \rightarrow \ldots (type(pn) \rightarrow X)\ldots)$ | X — новая типовая переменная

12) `f(`$x_1$`, `$x_2$`, …, `$x_n$`) | f имеет конкретный тип T -> R` =>

$type(x_1) = T$, $type(x_2) = R.From$, …,

$type(x_k | k <= n) = R.To.To\ldots.From$, пока R.To….To является функциональным типом. Если k < n, то ошибка.

**Замечание**: типы выражений, не содержащих подвыражений неопределенного типа, выводятся и проверяются обычным образом.

## 8.3 Унификация

После того, как для шаблона функции построено множество ограничений, необходимо проверить, что множество ограничений **совместно** — все ограничения на типы могут выполняться одновременно. К примеру, если множество ограничений $C$ имеет вид $C = \{..., X = Bool,\ X \in \{Int, Double\}, ...\}$ то есть несовместно, значит тело шаблона функции, для которого было построено это множество, содержит семантическую ошибку (т.к. не существует набор типов, удовлетворяющий множеству ограничений).

**Решением** множества ограничений назовем набор типов, удовлетворяющих этому множеству.

Для проверки непустоты множества решений **используется** унификация. Сначала приведем несколько определений.

Подстановка $\sigma$ называется ***менее конкретной*** (или ***более обобщенной***), чем подстановка $\sigma'$ и обозначается $\sigma \subseteq \sigma'$, если $\sigma' = \gamma \circ \sigma$ для некоторой подстановки $\gamma$.

Главным (наиболее общим) унификатором для множества ограничений *C* называется подстановка $\sigma$, удовлетворяющая *C*, если $\sigma \subseteq \sigma'$ для любой подстановки $\sigma'$, удовлетворяющей *C*.

Пример. Унификатором множества ограничений $C = \{X = Int, Y \to Bool = Z \to Bool, void \to Z = void \to Int\}$ является подстановка $\sigma = [X \mapsto Int, Y \mapsto Int, Z \mapsto Int]$.

В [2] Бенжамин Пирс приводит **алгоритм унификации** для множества ограничений в языке, являющемся подмножеством типизированного λ-исчисления. Множество ограничений состоит из уравнений вида *S* = *T*, где S, T — некоторые типы (то есть первичных ограничений):

```
unify*(C) =
     если C = ∅, то []
     иначе пусть {S = T} ∪ C′ = C, и тогда
          если S = T
               тогда unify*(C′)
          иначе если S = X и X∉FV(T)
               тогда unify*([X ↦ T]C′)∘[X ↦ T]
          иначе если T = X и X∉FV(S)
               тогда unify*([X ↦ S]C′)∘[X ↦ S]
          иначе если S = S1→S2 и T = T1→T2
               тогда unify*(C′ ∪ {S1 = T1, S2 = T2})
          иначе неудача
```

*FV(T)* — это множество типовых переменных в типе T. Запись *пусть* $\{S = T\} \cup C' = C$ означает «выберем ограничение *S* = *T* из набора ограничений *C* и обозначим множество оставшихся ограничений в *C* символом *C'*».

Доказывается, что в выбранной системе данный алгоритм всегда завершается. Он терпит неудачу, если получает на входе невыполнимый набор ограничений, а в противном случае возвращает главный унификатор.

Модифицируем алгоритм для нашего языка. В этом случае появляются вторичные ограничения ($S \in \{ T_i \}$). На самом деле, класс таких ограничений состоит из ограничений вида $X \in \{Int, Double\}$, где X — некоторый тип. Поэтому алгоритм унификации для языка PollyTL будет выглядеть следующим образом:

```
preunify(C) = unify*(C)

check(X) =
     если X = Int или X = Double
          то true
     иначе если X — конкретный тип
          то false
     иначе если X = S -> T
          то false
     иначе true

check(C = {X ∈ {Int, Double}}) =
     для каждого (X ∈ {Int, Double}) ∈ C
          если не check(X)
               то false
     true


unify(C) =
     r = preunify(C)
     если неудача
          то неудача
     иначе если check(C')
          то r
     иначе неудача
```

Если унификация множества ограничений шаблона завершилась неудачей, то шаблон функции содержит семантическую ошибку.

Успешная унификация, однако, не дает нам ответа на вопрос, можно ли вызывать шаблон функции от набора конкретных типов, она лишь показывает, что некоторый такой набор существует.

Простейшим, но *не самым эффективным* способом решения задачи о возможности инстанцировать шаблон набором конкретных типов при

вызове шаблонной функции *f(x1, …, xn)* является следующий: в исходный набор ограничений добавляем ограничения вида $X = T$, где X — типовая переменная-параметр шаблона, а T — соответствующий ей тип, выведенный по типам параметров. Унифицируем полученное множество ограничений. Если унификация удалась, вызов шаблона функции с данным набором параметров допустим. Заметим, что анализировать *тело* шаблона повторно не требуется.

Для анализа вызова шаблонной функции от набора типов, который содержит типовые переменные (то есть в случае вызова шаблонной функции внутри определения шаблона другой функции), этот алгоритм неприменим.

# 9. Руководство по развертыванию

Программная реализация компилятора представляет собой проект на языке C#. Проект находится в папке PollyTL.

Для запуска компилятора необходимо перейти в папку bin/Debug и исполнить файл PollyTLParser.exe, передав в качестве параметра имя текстового файла с программой на языке PollyTL. Также можно открыть в приложении MS VisualStudio проект PollyTLangProject.sln, настроить параметр командной строки и запустить проект.

# 10. Заключение

В результате работы был разработан модельный императивный язык программирования PollyTL с λ-функциями и шаблонами функций. Были реализованы front-end и middle-end части компилятора языка PollyTL. Компилятор выполняет синтаксический и семантический анализ программы, проводит проверку и вывод типов. Результатом работы компилятора является сообщение об ошибке в программе или вывод информации о типах выражений, запрошенных в операторе println.

Были исследованы некоторые алгоритмы обработки шаблонов функций, основанные на использовании множества ограничений и алгоритме унификации.

## Литература


1. Универсальные шаблоны (Руководство по программированию на C#) http://msdn.microsoft.com/ru-ru/library/512aeb7t.aspx
2. The Gardens Point Parser Generator (GPPG) http://plas.fit.qut.edu.au/gppg/
3. Бенжамин Пирс. Типы в языках программирования (перевод с английского). Раздел V, Глава 22 (реконструкция типов) http://dl.dropbox.com/u/132983/tapl.pdf

## **Приложение 1.** Грамматика языка PollyTL

```
/* ************* Основные блоки ************* */

codefile                                    // Кодовый файл
     : mainProgramBlock

mainProgramBlock                            // Программный блок
    : Declarations mainProgramFunc

mainProgramFunc                             // Основной исполняемый
раздел
    : MAIN Statements END


/* ************* Секция описаний ************* */

Declarations                          // Секция описаний
    : DeclarationsList
    | /* empty */

DeclarationsList                      // Список секций описаний
     : DeclarationsList DeclarationSection
     | DeclarationSection

DeclarationSection                    // Секция описания
    : FunctionDeclarationSection
     | VariableDeclarationSection
     | ConstantDeclarationSection

VariableDeclarationSection    // Объявление (описание) переменных
     : VAR ident ASSIGN expr SEMICOLUMN
     | TypeDescription VariableDefinitionsList SEMICOLUMN

VariableDefinitionsList       // Список объявлений/описаний переменных
     : VariableDefinitionsList COLUMN VariableDefinition
     | VariableDefinition

VariableDefinition            // Описание / объявление переменной
     : ident
     | ident ASSIGN expr

ConstantDeclarationSection          // Описание константы
     : CONST TypeDescription ident ASSIGN ConstExpr SEMICOLUMN

FunctionDeclarationSection          // Описание функции
    : ReturnType FUN ident TemplateParams LPAREN FunctionFor-
malParameters RPAREN Statements END
     | ReturnType FUN ident LPAREN FunctionFormalParameters
RPAREN Statements END
```

```
ReturnType                    // Возвращаемое значение функции
     : TypeDescription
     | /* empty */

FunctionFormalParameters      // Формальные параметры функции
     : FormalParametersList
     | ParameterDeclaration
     | /* empty */

TemplateParams                // Шаблонные параметры функции
     : OPEN_GENERIC IdentList RBRACKET

TypeWithIdentParameter   // Некоторый параметр вида <тип идентификатор>
     : TypeDescription ident


/* ************* Типы ************* */

TypeDescription               // Описание типа
     : StadardType
     | ArrowType

StadardType                   // Обычный тип
     : ident

ArrowType                // Функциональный тип (со стрелочкой)
     : InArrowType ARROW ArrowType
     | InArrowType ARROW InArrowType

InArrowType                   // Часть функционального типа
     : StadardType
     | LPAREN ArrowType RPAREN


/* ************* Выражения ************* */

ident                         // Идентификатор
    : ID

FullIndetifier                // Полный идентификатор (вместе с
точками, например, будет) TODO
     : ident

expr                          // Выражение TODO
     : expr Relation SimpleExpr
     | SimpleExpr

Relation                      // Отношение
     : EQ // ==
     | NE // !=
     | GE // >=
```

```
     | LE // <=
     | GT // >
     | LT // <

SimpleExpr                    // Простое выражение
     : SimpleExpr PlusOperator SignedTerm
     | SignedTerm

PlusOperator                  // Оператор "сложения"
     : PLUS
     | MINUS
     | OR // ||

SignedTerm                    // Слагаемое со знаком
     : term
     | PLUS term %prec UPLUS
     | MINUS term %prec UMINUS

term                          // Слагаемое
     : term MultOperator factor
     | factor

MultOperator                  // Оператор "умножения"
     : MULT
     | DIVIDE
     | AND // &&
     | DIV
     | MOD

factor                        // Множитель
     : FullIndetifier
     | NOT factor
     | BoolValue
     | NumericValue
     | CallFunction
     | ExplicitTemplateCallFunction
     | LambdaExpr
     | LPAREN expr RPAREN

BoolValue                     // Булево значение
     : TRUE
     | FALSE
NumericValue                  // Числовое выражение
     : INTNUM
     | DOUBLENUM

ConstExpr                     // Константное выражение
     : expr

LambdaExpr                    // Лямбда-выражение
     : LambdaParameters LAMBDA_ARROW LBRACE LambdaBody RBRACE
```

```
LambdaParameters              // Параметы лямбда-выражения
     : ident
     | TypeWithIdentParameter
     | LPAREN FormalParametersList RPAREN

LambdaBody                    // Тело лямбда-выражения
     : expr
     | StatementsList


/* ************ Списки ************ */

ExprList                      // Список выражений
     : ExprList COLUMN expr
     | expr

FormalParametersList          // Список формальных параметров
     : FormalParametersList COLUMN ParameterDeclaration
     | ParameterDeclaration COLUMN ParameterDeclaration

ParameterDeclaration          // Объявление (описание) параметра
     : TypeWithIdentParameter
     | ident

IdentList                     // Список идентификаторв
     : IdentList COLUMN ident
     | ident


/* ************ Секция предложений ************ */

Statements                    // Секция операторов
    : StatementsList
    | /* empty */

StatementsList                // Последовательность операторов
     : StatementsList Statement
     | Statement

Statement                     // Оператор
     : InternalDeclarations
     | Assignment
     | IfStatement
     | WhileStatement
     | EmptyStatement
     | CallFunction SEMICOLUMN
     | ExplicitTemplateCallFunction SEMICOLUMN
     | ReturnOperator

EmptyStatement                // Пустой оператор
     : SEMICOLUMN
```

```
InternalDeclarations          // Описания, возможные внутри программы
     : VariableDeclarationSection

Assignment                    // Присваивание
     : ident ASSIGN expr SEMICOLUMN

IfStatement                   // Оператор If
     : IF expr THEN StatementsList ElifIfStatementPart FI
     | IF expr THEN StatementsList ElifIfStatementPart ELSE
StatementsList FI

ElifIfStatementPart           // Набор Elif-операторов, может
быть пустым
    : ElifStatementsList
    | /* empty */

ElifStatementsList            // Список Elif-операторов
    : ElifStatementsList ElifStatement
    | ElifStatement

ElifStatement                 // Elif-выражение
    : ELIF expr THEN StatementsList

WhileStatement                // Оператор while
    : WHILE expr DO StatementsList ENDW

ReturnOperator                // Возвращение значения
     : RETURN SEMICOLUMN
     | RETURN expr SEMICOLUMN

CallFunction                  // Вызов функции
     : ident LPAREN FunctionFactParameters RPAREN

FunctionFactParameters        // Фактические параметры функции
     : ExprList
     | /* empty */

ExplicitTemplateCallFunction  // Вызов шаблона функции с явным
указанием типов параметров шаблона
     : ident OPEN_GENERIC TemplateTypesList RBRACKET LPAREN
FunctionFactParameters RPAREN


TemplateTypesList             // Список указаний типов парамет-
ров шаблона
     : TemplateTypesList COLUMN TemplateTypeDeclaration
     | TemplateTypeDeclaration
```

```
TemplateTypeDeclaration       // Указание типа шаблона
     : TypeDescription
     | ident ASSIGN TypeDescription
```