МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

САНКТ**-**ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ**,** МЕХАНИКИ И ОПТИКИ

Кафедра Информатики и прикладной математики

Курсовая работа

по предмету «Верификация моделей программ»

Выполнил:

студент гр. № Р4115

Назукин Д. Е.

Проверил :

Кореньков Ю. Д.

Санкт-Петербург

2017

**Оглавление**

[**Верификация синтаксических конструкций текстовых представлений данных** 3](#_Toc485066727)

[**Верификация алгоритмической последовательности модели программы** 6](#_Toc485066728)

[**Верификация формулируемости алгоритма в терминах моделей вычисления** 9](#_Toc485066729)

[**Верификация простых манипуляций со структурами данных с учетом модели типизации** 13](#_Toc485066730)

[**Верификации корректности гетерогенной программы методом интерпретации** 16](#_Toc485066731)

[**Верификация программ с учетом полиморфизма диспетчеризации** 20](#_Toc485066732)

[**Верификации программ с учетом параметрического полиморфизма типов.** 22](#_Toc485066733)

## **Верификация синтаксических конструкций текстовых представлений данных**

**Задачи**

Реализовать построение по исходному файлу с текстом синтаксического дерева с узлами, соответствующими правилам варианта, задающего язык для анализа. Вывести полученное дерево в файл.

**Описание работы**

Абстрактное синтаксическое дерево (АСД) — в [информатике](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) [конечное](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B5%D1%87%D0%BD%D0%BE%D0%B5_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE) помеченное ориентированное [дерево](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_(%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2)), в котором внутренние вершины сопоставлены (помечены) с [операторами](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) языка программирования, а листья — с соответствующими [операндами](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D0%BD%D0%B4). Таким образом, листья являются пустыми операторами и представляют только переменные и константы.

Синтаксические деревья используются в [парсерах](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7" \o "Синтаксический анализ) для [промежуточного представления](https://ru.wikipedia.org/w/index.php?title=%D0%9F%D1%80%D0%BE%D0%BC%D0%B5%D0%B6%D1%83%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5&action=edit&redlink=1) программы между [деревом разбора](https://ru.wikipedia.org/w/index.php?title=%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D1%80%D0%B0%D0%B7%D0%B1%D0%BE%D1%80%D0%B0&action=edit&redlink=1) (конкретным синтаксическим деревом) и [структурой данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), которая затем используется в качестве внутреннего представления в [компиляторе](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80) или [интерпретаторе](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80) [компьютерной программы](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) для оптимизации и генерации кода. Возможные варианты подобных структур описываются абстрактным синтаксисом.

**Аспекты реализации**

Для реализации построения абстрактного синтаксического дерева была использована библиотека ANTLR4 для языка Java.

ANTLR — [генератор](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80_%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80%D0%BE%D0%B2) [нисходящих](https://ru.wikipedia.org/wiki/%D0%9D%D0%B8%D1%81%D1%85%D0%BE%D0%B4%D1%8F%D1%89%D0%B8%D0%B9_%D1%81%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7) [анализаторов](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D1%81%D0%B5%D1%80) для формальных языков. ANTLR преобразует [контекстно-свободную грамматику](https://ru.wikipedia.org/wiki/LL-%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) в виде [РБНФ](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D1%88%D0%B8%D1%80%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%D1%84%D0%BE%D1%80%D0%BC%D0%B0_%D0%91%D1%8D%D0%BA%D1%83%D1%81%D0%B0_%E2%80%94_%D0%9D%D0%B0%D1%83%D1%80%D0%B0) в программу на [C++](https://ru.wikipedia.org/wiki/C%2B%2B), [Java](https://ru.wikipedia.org/wiki/Java" \o "Java), [C#](https://ru.wikipedia.org/wiki/C_Sharp), [Python](https://ru.wikipedia.org/wiki/Python" \o "Python), [Ruby](https://ru.wikipedia.org/wiki/Ruby" \o "Ruby). Используется для разработки [компиляторов](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80%D1%8B), [интерпретаторов](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80) и [трансляторов](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80).

Ниже приведена часть грамматики:



После описания грамматики библиотека сгенерирует необходимые классы для парсинга текста и построения AST дерева.



**Результаты**

Вход программы файл с текстом:

function sto()  
 sto = 0;  
 do  
 sto = sto+1;  
 loop while (sto < 100)  
end function

Полученное дерево:

Source  
 SourceItem  
 FuncDef  
 FuncSignature  
 Identifier: sto  
 ListArgDef  
 ExpressionStatement  
 AssignExpr  
 PlaceExpr  
 Identifier: sto  
 LiteralExpr: 0  
 DoStatement  
 Type: while  
 ExpressionStatement  
 AssignExpr  
 PlaceExpr  
 Identifier: sto  
 BinaryExpr  
 PlaceExpr  
 Identifier: sto  
 BinOp: +  
 LiteralExpr: 1  
 BracesExpr  
 BinaryExpr  
 PlaceExpr  
 Identifier: sto  
 BinOp: <  
 LiteralExpr: 100

**Вывод**

В данной главе реализовано построение по исходному файлу с текстом синтаксического дерева с узлами, соответствующими правилам варианта, задающего язык для анализа. Выведено полученное дерево в файл.

## **Верификация алгоритмической последовательности модели программы**

**Задачи**

Используя анализатор от первого задания реализовать программу, принимающую в качестве входных аргументов командной строки список имен N входных файлов и путь к выходной директории. Для заданных входных файлов с текстами на соответствующем варианту языке программа должна формировать в выходной директории выходные файлы, содержащие представление графа потока управления, описывающего последовательность выполнения операций, заданных инструкциями и выражениями соответствующей части исходного текста. В той же директории должен формироваться дополнительный файл, содержащий представление графа вызовов для всей совокупности входных файлов, описывающее отношения между функциями/методами входных файлов в плане обращению их друг к другу по именам.

**Описание работы**

Граф потока управления ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) control flow graph, CFG) — в [теории компиляции](https://ru.wikipedia.org/w/index.php?title=%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%86%D0%B8%D0%B8&action=edit&redlink=1) — множество всех возможных путей исполнения программы, представленное в виде [графa](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)" \o "Граф (математика)).

В графе потока управления каждый узел (вершина) графа соответствует [базовому блоку](https://ru.wikipedia.org/wiki/Basic_block) — прямолинейному участку кода, не содержащему в себе ни операций передачи управления, ни точек, на которые управление передается из других частей программы. Имеется лишь два исключения:

* точка, на которую выполняется переход, является первой инструкцией в базовом блоке;
* базовый блок завершается инструкцией перехода.

Направленные дуги используются в графе для представления инструкций перехода. Также, в большинстве реализаций добавлено два специализированных блока:

* входной блок, через который управление входит в граф;
* выходной блок, который завершает все пути в данном графе.

**Аспекты реализации**

Граф потока управления строится по абстрактному синтаксическому дереву. Для обхода дерева был реализован паттерн Посетитель (англ. visitor). В графе есть либо обычные узлы, либо узлы в которых происходит разветвление потока. Разветвление начинается в следующих случаях:

* Условие if
* Цикл while
* Цикл Do while
* Цикл Do until

**Результаты**

Вход программы файл с текстом:

function calc(A as long, B as long, Op as char)  
 if (Op == '+') then  
 calc = sum(A, B);  
 end if  
 if (Op == '-') then  
 calc = minus(A,B);  
 end if  
end function  
  
function minus(A as long, B as long)  
 minus = (A-B);  
end function  
  
function sum(A as long, B as long)  
 sum = A+B;  
end function

Результат:

Файл calc.txt

start -> 0  
0 t-> 1  
1 -> 2  
2 t-> 3  
3 -> end  
2 f-> end  
0 f-> 2  
  
0:(Op=='+')  
1:calc=sum(A,B)  
2:(Op=='-')  
3:calc=minus(A,B)

Файл sum.txt

start -> 0  
0 -> end  
  
0:sum=A+B

Файл minus.txt

start -> 0  
0 -> end  
  
0:minus=(A-B)

Файл funcsCFG.txt

calc->sum  
calc->minus

**Вывод**

В данной главе реализовано построение графа потока управления для каждой функции, а также графа вызовов для всей совокупности входных файлов, описывающее отношения между функциями/методами входных файлов в плане обращению их друг к другу по именам.

## **Верификация формулируемости алгоритма в терминах моделей вычисления**

**Задачи**

Дополнить программу из второго задания построением линейного кода определённого в соответствии с вариантом задания (регистровый байт код для языка с динамической типизацией) вида на основе имеющегося графа потока управления. Полученный линейный код для функций или методов вывести в мнемонической форме в файлы в той же директории.

**Описание работы**

В регистровой реализации виртуальной машины структура данных, в которую помещаются операнды, основана на регистрах процессора. При этом не требуются операции PUSH или POP, но инструкции должны явно содержать адреса (регистры) в которых содержатся операнды. То есть, операнды для инструкций, в отличии от стековой модели, указываются явно.  Например, операция сложения в регистровой виртуальной машине выглядит приблизительно так:

|  |  |
| --- | --- |
|  | ADD R1, R2, R3; # складывает содержимое R1 и R2, результат  заносит в R3 |

За счет отсутствия операций POP и PUSH команды в регистровой виртуальной машине выполняются быстрее аналогичных команд стековой виртуальной машины.

Другое преимущество регистровой модели в том, что она позволяет провести оптимизацию, которая не может быть выполнена при стековом подходе. Например, несколько раз встречающееся выражение при регистровом подходе может быть вычислено лишь однажды и сохранено в регистре для последующего использования, что экономит время необходимое для пересчета выражения. Но с другой стороны в среднем инструкция регистровой машины длиннее чем в стековой машине, так как в ней требуется явное указание операндов.

**Аспекты реализации**

Регистровый байт-код строится по абстрактному синтаксическому дереву. Для обхода дерева был реализован паттерн Посетитель (англ. visitor). Заходя в узел дерева определенного вида строится свой код.

Например, следующим образом строится код для узла бинарной операции:

**public** Void visitBinaryExpr(BinaryExprContext ctx) {  
 String binOp = ctx.binOp().getText();  
 **int** firstReg;  
 **switch** (binOp) {  
 **case "+"**:  
 visit(ctx.expr(0));  
 firstReg = **lastRegistrNum**;  
 visit(ctx.expr(1));  
 gen(**"ADD"**, firstReg, **lastRegistrNum**, firstReg);  
 **lastRegistrNum** = firstReg;  
 **break**;  
 **case "-"**:  
 visit(ctx.expr(0));  
 firstReg = **lastRegistrNum**;  
 visit(ctx.expr(1));  
 gen(**"MINUS"**, firstReg, **lastRegistrNum**, firstReg);  
 **lastRegistrNum** = firstReg;  
 **break**;  
 **case "\*"**:  
 visit(ctx.expr(0));  
 firstReg = **lastRegistrNum**;  
 visit(ctx.expr(1));  
 gen(**"MULT"**, firstReg, **lastRegistrNum**, firstReg);  
 **lastRegistrNum** = firstReg;  
 **break**;  
 **case "/"**:  
 visit(ctx.expr(0));  
 firstReg = **lastRegistrNum**;  
 visit(ctx.expr(1));  
 gen(**"DIV"**, firstReg, **lastRegistrNum**, firstReg);  
 **lastRegistrNum** = firstReg;  
 **break**;  
 **case "%"**:  
 visit(ctx.expr(0));  
 firstReg = **lastRegistrNum**;  
 visit(ctx.expr(1));  
 gen(**"MOD"**, firstReg, **lastRegistrNum**, firstReg);  
 **lastRegistrNum** = firstReg;  
 **break**;  
 **case "<"**:  
 visit(ctx.expr(0));  
 firstReg = **lastRegistrNum**;  
 visit(ctx.expr(1));  
 gen(**"LESS"**, firstReg, **lastRegistrNum**, firstReg);  
 **lastRegistrNum** = firstReg;  
 **break**;  
 **case ">"**:  
 visit(ctx.expr(0));  
 firstReg = **lastRegistrNum**;  
 visit(ctx.expr(1));  
 gen(**"LARGER"**, firstReg, **lastRegistrNum**, firstReg);  
 **lastRegistrNum** = firstReg;  
 **break**;  
 **case "=="**:  
 visit(ctx.expr(0));  
 firstReg = **lastRegistrNum**;  
 visit(ctx.expr(1));  
 gen(**"EQUAL"**, firstReg, **lastRegistrNum**, firstReg);  
 **lastRegistrNum** = firstReg;  
 **break**;  
 **case "||"**:  
 visit(ctx.expr(0));  
 firstReg = **lastRegistrNum**;  
 visit(ctx.expr(1));  
 gen(**"OR"**, firstReg, **lastRegistrNum**, firstReg);  
 **lastRegistrNum** = firstReg;  
 **break**;  
 **case "&&"**:  
 visit(ctx.expr(0));  
 firstReg = **lastRegistrNum**;  
 visit(ctx.expr(1));  
 gen(**"AND"**, firstReg, **lastRegistrNum**, firstReg);  
 **lastRegistrNum** = firstReg;  
 **break**;  
 **default**:  
 **throw new** RuntimeException(**"Unexpected binary operator"** + binOp);  
 }  
 **return null**;  
}

**Результаты**

Вход программы файл с текстом:

function calc(A as long, B as long, Op as char)  
 if (Op == '+') then  
 calc = sum(A, B);  
 end if  
 if (Op == '-') then  
 calc = minus(A,B);  
 end if  
end function

Результат:

.methodSignature  
MethodSignature{funcName='calc', argCount=3, argsType=[long, long, char], returnType=undefined}  
.funcs  
0:sum  
1:minus  
.vars  
0:calc  
1:A  
2:B  
3:Op  
.consts  
0:Const{constName=''+'', type=char}  
1:Const{constName=''-'', type=char}  
.programm  
0: PUSHVAR 3 0   
1: PUSHCONST 0 1   
2: EQUAL 0 1 0   
3: JMPFALSE 0 9   
4: PUSHVAR 1 0   
5: PUSHVAR 2 1   
6: CALL 0 0 0 1   
7: LOADVAR 0 0   
8: JMP 9   
9: PUSHVAR 3 1   
10: PUSHCONST 1 2   
11: EQUAL 1 2 1   
12: JMPFALSE 1 18   
13: PUSHVAR 1 1   
14: PUSHVAR 2 2   
15: CALL 1 1 1 2   
16: LOADVAR 1 0   
17: JMP 18   
18: END

Описание функции представляет собой:

1. Сигнатура функции
2. Вызываемые внешние функции
3. Локальные переменные
4. Константы
5. Байт-код

**Вывод**

В данной главе дополнена программа из второго задания построением регистрового байт-кода для языка с динамической типизацией на основе имеющегося графа потока управления. Полученный линейный код для функций выведен в мнемонической форме в файлы.

## **Верификация простых манипуляций со структурами данных с учетом модели типизации**

**Задачи**

Создать библиотеку, позволяющую сохранять в бинарный файл и загружать из него набор закодированных линейным кодом в соответствии с предыдущим заданием функций или методов. Изменить имеющуюся программу так, чтобы она, кроме входных файлов и имени выходной директории, принимала через аргументы командной строки имя для выходного файла с бинарным представлением, формируемого с помощью созданной библиотеки. Дополнить входной язык возможностью определения пользовательских типов данных, обладающих набором полей и методов.

**Описание работы**

Пользовательские типы данных позволяют создавать пользователю собственные типы данных (классы) отличных от стандартных. Класс может обладать набором полей и методов с определенными модификаторами доступа (*public*, *private*). *Public* означает что метод или поле доступно к доступу вне класса, *private* же означает что метод или поле будет доступно только внутри реализации класса. Каждый класс должен обладать конструктором, который инициализирует новый член класса. Члены класса создаются с помощью оператора *new.*

**Аспекты реализации**

Для сохранения в бинарный файл было использовано Java Serialization API. Класс сериализуемого объекта должен реализовывать интерфейс *java.io.Serializable:*

**public class** MethodDefinition **implements** Serializable

Запись и чтение из файла происходит следующим образом:

**public void** printBinCodes(Map<MethodSignature, MethodDefinition> binFuncs, Map<String, ClassDefinition> binClasses, String outputPath) **throws** IOException {  
 File file = **new** File(outputPath);  
 file.getParentFile().mkdirs();  
 **try** (ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream(file))) {  
 List<Object> list = **new** ArrayList<>();  
 list.add(binFuncs);  
 list.add(binClasses);  
 oos.writeObject(list);  
 oos.flush();  
 } **catch** (Exception e) {  
 **throw** e;  
 }  
}

**public** List<Object> readBinCodes(String path) **throws** IOException, ClassNotFoundException {  
 **try** (ObjectInputStream oin = **new** ObjectInputStream(**new** FileInputStream(path))) {  
 **return** (List<Object>) oin.readObject();  
 } **catch** (Exception e) {  
 **throw** e;  
 }  
}

Для хранения структуры класса используется следующий класс:

**public class** ClassDefinition **implements** Serializable {  
 **private** String **className**;  
  
 **private** Map<String, Integer> **fields** = **new** LinkedHashMap<>();  
 **private** Map<String, Modificator> **fieldsModificator** = **new** LinkedHashMap<>();  
  
 **private** Map<MethodSignature, MethodDefinition> **functions** = **new** HashMap<>();  
 **private** Map<MethodSignature, Modificator> **functionsModificator** = **new** HashMap<>();

Он содержит имя класса, поля с модификаторами и методы с модификаторами.

**Результаты**

Вход программы файл с текстом:

class User  
 private id, name  
  
 public function getName()  
 getName = name;  
 end function  
  
 public function setName(\_name as string)  
 name = \_name;  
 end function  
  
 public function getId()  
 getId = id;  
 end function  
  
 public function setId(newId as long)  
 id = newId;  
 end function  
  
 public function New(\_id as long, \_name as string)  
 id = \_id;  
 name = \_name;  
 end function  
  
end class

Мнемоническое описание класса:

---Class---  
User  
---fields---  
0 : id - PRIVATE  
1 : name - PRIVATE  
---functions---  
PUBLIC :

.methodSignature  
MethodSignature{funcName='setId', argCount=1, argsType=[undefined], returnType=undefined}  
.funcs  
.vars\_count  
2  
.consts  
.programm  
0: PUSHVAR 1 0   
1: LOADCLASSVAR 0 0   
2: END   
  
PUBLIC :   
 .methodSignature  
MethodSignature{funcName='getId', argCount=0, argsType=[], returnType=undefined}  
.funcs  
.vars\_count  
1  
.consts  
.programm  
0: PUSHCLASSVAR 0 0   
1: LOADVAR 0 0   
2: END   
  
PUBLIC :   
 .methodSignature  
MethodSignature{funcName='getName', argCount=0, argsType=[], returnType=undefined}  
.funcs  
.vars\_count  
1  
.consts  
.programm  
0: PUSHCLASSVAR 1 0   
1: LOADVAR 0 0   
2: END

PUBLIC :   
 .methodSignature  
MethodSignature{funcName='setName', argCount=1, argsType=[string], returnType=undefined}  
.funcs  
.vars\_count  
2  
.consts  
.programm  
0: PUSHVAR 1 0   
1: LOADCLASSVAR 0 1   
2: END

PUBLIC :

.methodSignature  
MethodSignature{funcName='New', argCount=2, argsType=[long, string], returnType=undefined}  
.funcs  
.vars\_count  
3  
.consts  
.programm  
0: PUSHVAR 1 0   
1: LOADCLASSVAR 0 0   
2: PUSHVAR 2 1   
3: LOADCLASSVAR 1 1   
4: END

**Вывод**

В данной главе реализована возможность вывода программы в бинарном виде. Добавлена поддержка пользовательских типов данных.

## **Верификации корректности гетерогенной программы методом интерпретации**

**Задачи**

Воспользовавшись созданной в предыдущем задании библиотекой, создать программу, принимающую в качестве аргумента командной строки имя файла, содержащего бинарное представление кода программы на исходном языке, загружающую этот файл и выполняющую сохранённую в нём программу. Информацию о точке входа получать аргументом командной строки или из информации в бинарном файле. Взаимодействие загруженного и выполняемого кода с остальной компьютерной системой реализовать через поддержку импортирования произвольных, задаваемых во входном коде, функций внешних DLL, используя функции ввода-вывода WinApi.

**Описание работы**

В данной работе будет реализован интерпретатор байт-кода из задания № 3. Интерпретатор (interpreter) — [программа](http://megabook.ru/article/%d0%9f%d1%80%d0%be%d0%b3%d1%80%d0%b0%d0%bc%d0%bc%d0%b0%20(%d0%ba%d0%be%d0%bc%d0%bf%d1%8c%d1%8e%d1%82%d0%b5%d1%80%d1%8b%20%d0%b8%20%d0%b8%d0%bd%d1%82%d0%b5%d1%80%d0%bd%d0%b5%d1%82)) или техническое средство, выполняющее интерпретацию, а также вид транслятора, осуществляющего пооперационную (покомандную) обработку и выполнение исходной программы или запроса. В отличие от компилятора, который осуществляет трансляцию всей программы высокого уровня в машинные коды один раз без ее выполнения (создает объектную программу), интерпретатор транслирует исходную программу команда за командой каждый раз при выполнении и не создает объектного модуля.

Также будет реализована поддержка импортирования произвольных, задаваемых во входном коде, функций внешних DLL. DLL — это библиотека содержащая код и данные, которые могут использоваться более чем одной программой одновременно.

**Аспекты реализации**

Интерпретатор работает следующим образом:

1. Считывает текущую команду
2. Выполняет текущую команду
3. Переходит к следующей команде. Затем шаг 1, пока не достигнется завершающая команда.

**while** (**true**) {  
 Command command = **commands**.get(**currentCommand**);  
 **if** (command.getName().equals(**"END"**)) {  
 **break**;  
 } **else** {  
 executeCommand(command);  
 }  
}

**private void** executeCommand(Command command) {  
 **switch** (CommandType.*valueOf*(command.getName())) {  
 **case *ADD***:  
 addCommand(command.getArgs());  
 **currentCommand**++;  
 **break**;  
 **case *MINUS***:  
 minusCommand(command.getArgs());  
 **currentCommand**++;  
 **break**;

. . .

Для поддержки импортирования внешний функций из DLL была написана *С* функция которая вызывается через *JNI*.

**public native** AbstractValue callNativeFunc(String lib, String funcName, AbstractValue[] args, String[] argsType, String returnType);

Для выполнения произвольных функции из DLL по их сигнатуре была использована библиотека *libffi.* Зная адрес функции (через *GetProcAddress*), типы аргументов (массив *ffi\_argTypes*) и тип возвращаемого значения (*c\_retType*), можно вызвать эту функцию передав указатели на аргументы в виде массива *void\* (values).* Результат выполнения будет в переменной *rc.*

ffi\_arg rc; *// return value*

**if** (ffi\_prep\_cif(&cif, FFI\_DEFAULT\_ABI, length, c\_retType, ffi\_argTypes) == FFI\_OK) {  
  
 ffi\_call(&cif, FFI\_FN(GetProcAddress(dllHandle, funcC)), &rc, values);  
} **else** {  
 **return** NULL;  
}

**Результаты**

Вход программы файл с текстом:

declare function CreateFile(lpFileName as string,  
 dwDesiredAccess as int,  
 dwShareMode as int,  
 lpSecurityAttributes as int,  
 dwCreationDisposition as int,  
 dwFlagsAndAttributes as int,  
 hTemplateFile as int) as int  
 lib "kernel32.dll" alias "CreateFileA"  
  
declare function WriteFile(hFile as int,  
 lpBuffer as string,  
 nNumberOfBytesToWrite as int,  
 lpNumberOfBytesWritten as int,  
 lpOverlapped as int) as int  
 lib "kernel32.dll"  
  
declare function CloseHandle(hFile as int) as int lib "kernel32.dll"  
  
declare function lstrlen(hFile as string) as int lib "kernel32.dll" alias "lstrlenA"

function print(text as string, handle)  
 WriteFile(handle, text, lstrlen(text), 0, 0);  
end function  
  
function main()  
 f = CreateFile("out.txt", 0x40000000, 0, 0, 2, 0, 0);  
  
 a = new User(3, "Ivan");

print("User= " + a.toString() + "\n", f);

CloseHandle(f);  
  
 main = 0;  
end function

class User  
 private id, name

public function New(\_id as long, \_name as string)  
 id = \_id;  
 name = \_name;  
end function

public function toString()  
 toString = "id: " + getId() + " name: " + name;  
end function

end class

В результате будет создан файл out.txt и в него записан результат:

User= id: 3 name: Ivan

**Вывод**

В данной главе реализована программа (интерпретатор), принимающая в качестве аргумента командной строки имя файла, содержащего бинарное представление кода программы на исходном языке, загружающую этот файл и выполняющую сохранённую в нём программу.

Также реализовано импортирование произвольных внешних функций из DLL.

## **Верификация программ с учетом полиморфизма диспетчеризации**

**Задачи**

Добавить в разработанный за предыдущие задания программный комплекс поддержку перегрузки функций/методов и поддержку полиморфизма подтипов с переопределением реализаций логики (опционально виртуальные методы).

**Описание работы**

Под перегрузкой функции понимается, определение нескольких функций (две или больше) с одинаковым именем, но различными параметрами. Наборы параметров перегруженных функций могут отличаться порядком следования, количеством, типом. Таким образом перегрузка функций нужна для того, чтобы избежать дублирования имён функций, выполняющих сходные действия, но с различной программной логикой.

В объектно-ориентированном программировании полиморфизм подтипов (или полиморфизм включения) представляет собой концепцию в теории типов, предполагающую использование единого имени (идентификатора) при обращении к объектам нескольких разных классов, при условии, что все они являются подклассами одного общего надкласса (суперкласса). Полиморфизм подтипов состоит в том, что несколько типов формируют подмножество другого типа (их базового класса) и потому могут использоваться через общий интерфейс.

**Аспекты реализации**

Для реализации перегрузки функций на этапе компиляции фиксируется только название вызываемой функции, а конкретная функция уже выбирается во время выполнения кода. Это сделано так, потому что при динамической типизации на этапе компиляции мы не знаем типы аргументов, которые передадутся в функцию, и соответственно не знаем, какая из перегруженных функций должна быть вызвана. Таким образом, типы аргументов в сигнатуре функции фиксируют типы аргументов, передаваемых в эту функцию.

**Результаты**

Вход программы файл с функциями (приведены только основные функции):

public function sum(A as long, B as long, f)  
 sum = A+B;  
 print("long sum= " + sum + "\n", f);  
end function  
  
public function sum(A as string, B as string, f)  
 sum = A+B;  
 print("string sum= " + sum + "\n", f);  
end function  
  
public function sum(A as bool, B as bool, f)  
 sum = A||B;  
 print("bool sum= " + sum + "\n", f);  
end function

function main()  
 f = CreateFile("out.txt", 0x40000000, 0, 0, 2, 0, 0);  
  
 a.sum("55","44",f);  
 a.sum(55,44,f);  
  
 a.sum(true,false,f);

main=0;  
end function

В результате будет создан файл out.txt и в него записан результат:

string sum= 5544  
long sum= 99  
bool sum= true

**Вывод**

В данной главе реализована добавлена в разработанный за предыдущие задания программный комплекс поддержка перегрузки функций.

## **Верификации программ с учетом параметрического полиморфизма типов.**

**Задачи**

Реализовать в подготовленном программном комплексе поддержку параметрического полиморфизма через обобщения.

**Описание работы**

Параметрический полиморфизм (parametric polymorphism) — это возможность определения обобщённых структур данных и функций, поведение которых не зависит от типов значений, которыми они оперируют. В случае типов данных значения произвольных типов могут тем или иным образом использоваться внутри контейнеров (непосредственно содержаться в контейнерах, либо содержимое контейнеров будет иметь какую-либо зависимость от таких произвольных типов). В случае функций именно поведение функции не зависит от типов, передаваемых таким функциям значений в качестве входных параметров.

**Аспекты реализации**

В случае динамической типизации обобщение на уровне структур данных не требует дополнительной реализации, т.к. поля класса (как хранилища для значений) на этапе компиляции и так не имеют типа и могут хранить в себе значения любых типов.

Обобщенные функции были реализованы следующим способом. Для того, чтобы указать, что аргумент функции является обобщенным, достаточно не указывать тип аргумента. Например:

public function less(A, B)  
 less = A<B;  
end function

В данном примере функция может быть вызвана с аргументами любых типов.

Соответственно, с учетом этих изменений алгоритм поиска функции для вызова следующий:

1. Искать функцию, сигнатура которой строго соответствует типам аргументов.
2. Если не найдена на шаге 1, искать нативную функцию по названию и кол-ву аргументов.
3. Если не найдена на шаге 2, искать функцию с учетом обобщения аргументов.

**Результаты**

Вход программы файл с функциями (приведены только основные функции):

public function less(A, B, f)  
 less = A<B;  
 print("result= " + less + "\n", f);  
end function

function main()  
 f = CreateFile("out.txt", 0x40000000, 0, 0, 2, 0, 0);  
  
 less(44, 55, f);

less("55", "44", f);

less(true, false, f);

main=0;  
end function

В результате будет создан файл out.txt и в него записан результат:

result= true  
result= false  
result= false

**Вывод**

В данной главе реализована в подготовленном программном комплексе поддержка параметрического полиморфизма через обобщения.