БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра информатики

Факультет НиДО

Специальность ИиТП

Практическая работа № 1

по дисциплине «Методы трансляции»

Выполнил студент: Дегтярев А.А.

группа 393551

Зачетная книжка № 902021-26

Минск 2017

### Вариант 7

### ИПР №1

**Цель:** Ознакомление студентов с разработкой языка программирования.

**Общая постановка задачи:**

Разработка языка программирования с заведомо определенными свойствами и создание простого компилятора, результатом работы которого является создание байт-кода

**Методические указания**:

Необходимо определить алфавит, синтаксис, грамматику простейшего языка программирования, разработать простой компилятор, результатом работы которого является создание байт-кода.

На основе имеющейся информации о грамматике языка разработать структуру байт-кода, который будет генерироваться компилятором. Необходимо при этом учитывать, что эта структура должна быть достаточно проста, чтобы в дальнейшем, при интерпретации байт-кода, затрачивался минимум временных и вычислительных ресурсов.

Разработать программный код, реализующий функцию генерации байт-кода и записи его в файл. Создать тестовое приложение.

**Подмножество языка включает**:

данные типа **int, float, struct** из элементов указанных типов;

* инструкции описания переменных;
* операторы присваивания в любой последовательности;
* операции **+, – , \*, = =, != , <, >** .

Из задания я предположил, что необходимо создание функции генерации байт-кода(как промежуточного (**intermediate**) языка); Так как подразумевается дальнейшая его интерпретация,другими программами(виртуальной машиной, или специализированным транслятором в машинный язык);

Создание intermediate языка позволит дальше выполнить операции оптимизации кода, упростит само устройство компилятора(не нужно будет делать полный нативный компилятор для каждой новой машины)(фронтенд остается прежним для всех конечных переводчиков в машинный код)

Возможно представление в разных вариантах промежуточного кода, в высокоуровневом(где язык вполне похож на высокоуровневый msil) или низкоуровневый вроде asm. Есть универсальный трех-адрессовый код(используется 3 регистра)  
  
Java bytecode использует достаточно высокоуровневые конструкции, тут есть таблица значений opcodes <https://en.wikibooks.org/wiki/Java_Programming/Byte_Code>

Для обхода графа ActivationTree, я использовал гибридный подход, в виде прохода по списку, элементами которого являются небольшие AST-деревья.

Проход по списку линейный, по деревьям рекурсивный, начиная с заглавной ноды;

Проход по дереву реализован в методе класса ноды дерева: toByteCode();

Для полноценной работы не хватает механизма управления памятью и выборки данных из регистров. Для этого необходимо создать какое-то подобие RTTI таблицы, которая бы поддерживалась в ходе генерации промежуточного кода;

|  |
| --- |
| //ipr1 TODO:  //int float struct  //type id;  //type id = literal;  //lop = rop  //+ - / \* == != < >  //AST-TREE has such order [OP|OPERAND1|OPERAND2|RES]  ////COMPILER BACK-END  public void IntermediateCodeGen(){  auto outputFile = File(outputFilePath,"w");  ActivationNode[] codeDAG;  //generarte intermediate lang  auto memOffset = 0;  //DAG  foreach(node;codeDAG){  outputFile.write(node.toByteCode(false));  }  }  public class ActivationNode{  public string type;  public string content;  public ActivationNode op;  public ActivationNode loperand;  public ActivationNode roperand;  //hosts in AST MANNER TREE INSIDE  string toByteCode(bool lop){  //end node  if(!op && !loperand && !roperand){  switch(type){  case "operator":  return getOpCode(content);  case "id":  if(lop){  //get address to move  return "store\_0" ~ "" ~ content;  }else{  //TODO: we need to fetch runtime table to get proper reg  return "load\_1" ~ "" ~ content;  }  case "literal":  if(lop){  //no assignment to const  return "ERROR!";  }else{  return "const\_2" ~ "" ~ content;  }  default:  assert(0);  }  }else{  //traverse tree  return roperand.toByteCode(false) ~ "\n" ~ op.toByteCode(false) ~ "\n" ~ loperand.toByteCode(true);  }  //here we can make some code opts but we need runtime data  }  static string getOpCode(string cont){  switch(cont){  case "+":  return "add";  case "-":  return "add neg";  case "\*":  return "mul";  case "/":  return "div";  case "==":  return "cmp";  case "!=":  return "cmp";  case "<":  return "cmp";  case ">":  return "cmp";  default:  assert(0);  }  }  } |