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

Лабораторная работа №3 (8 вариант)

по предмету «Верификация моделей программ»

Выполнил:

студент гр. № Р4115

Назукин Д. Е.

Проверил :

Кореньков Ю. Д.

Санкт-Петербург

2017

**Цели**

Освоение верификации 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 ходе работы 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.