Assignment 2 (CS335A)

❖ Group Details

Name	Roll No.	email-ID
Anupriy	150121	anupriy@iitk.ac.in
Deepanshu Bansal	150219	dbansal@iitk.ac.in
Jaskirat Singh	150302	jaskis@iitk.ac.in

❖ 3AC requirements:

- The jump/goto statement must contain a label to go to. Simply stating the line number(as in goto: 15) in the three address code won't work.
- We added the print and scan commands which print and scan the value of the variable respectively. Ex: the cmd 'print a ' will print the value of a on the console.
- Print is done on stdout and scan is done from stdin.

Command	IR 3AC Syntax	Examples
Scan	<lineNo>,scan,<id>	1,scan,a
Print	<lineNo>,print,<[id | num]>	2,print,a
BinOps	<lineNo>,<op>,<id>,<[id | num]>,<[id | num]>	3,+,a,b,c
UniOp	<lineNo>,-,<id>,<id>	4,-,a,a
Copy	<lineNo>,=,<id>,<[id | num]>	5,=,a,b
CondJump	<lineNo>,ifgoto,<relop>,<[id | num]>,<[id | num]>,<label>	6,ifgoto,leq,a,50,foo
UncondJump	<lineNo>,goto,<label>	7,goto,foo
Procedure	<lineNo>,call,<label>	8,call,foo
Label	<lineNo>,<label>	9,foo
Return	<lineNo>,ret	10,ret

❖ Project Structure

- ➢ **“global.h” -** This file contains all the declarations and definitions used in the whole project. It contains classes for 3AC Instructions, SymbolTable, RegisterDescriptor, BasicBlock and many others.
- ➢ **“loadData.cpp” -** It loads the IR file into the data structure for instructions (3AC).
- ➢ **“basicBlocks.cpp” -** It contains all the functions related to basic blocks like find leaders for each BB then assigning all BB’s their number, liveliness and next use of variables in BB and some other helper functions.
- ➢ **“allocateRegister.cpp” -** This file contains acts as a register allocator for our project. It handles all the business related to allocating registers.
- ➢ **“mainDriver.cpp” -** This file serves as a driver to invoke all functions properly and finally to get the desired result.
- ➢ **“generateCode.cpp” -** It translates each BB into x86 Assembly code and has all the syntax to handle each instruction, it uses above files and functions to generate the correct working assembly code.

❖ Working:

- ➢ **mainDriver.cpp** is the main driver program for the compiler.
- ➢ It loads the data as three address code instruction using the function **loadData()** mentioned in the file **loadData.cpp**
- ➢ Next, leaders of basic blocks are computed using **findLeaders()** in file **basicblocks.cpp**. Using position of the leaders, we assign the basic blocks using the function **assignBasicBlocks()** in **basicBlocks.cpp**
- ➢ After assigning the basic blocks, we find for each instruction liveliness and nextuse of variables present in that instruction, iterating over each basic block separately. This is done using the function **assignIsLiveAndNextUseEachBB()** defined in file **basicBlocks.cpp**.
- ➢ Also, we are assigning labels to each basic blocks (realising the fact that each and every basic block is assigned to a unique label)
- ➢ After finding the basic blocks, **generateCode()** function is called written in the file **generateCode.cpp**. This function iterates over each basic block and further iterates over each instruction in the basic block sequentially.
- ➢ Given that each variable was global, we are using the **.data** feature of x86 to allocate memory for each variable. This feature allowed us to create a

memory unit whose name was same as that of the variable. This is done before generating the code for each instructions.

- ➢ At each instruction, **allocateRegister()** is called which allocates the register to required variables which is used in generation of the code. The **allocateRegister()** function is same as getreg function which was discussed in the class, implemented in the file allocateRegister.cpp
- ➢ Having registers allocated, we generate the code for each instruction.
- ➢ Finally, generated code in x86 is printed in stdout using the function **printAssemblyCode()** written in **helper.cpp**.

❖ Assumptions:

As mentioned in the problem statement, we made the following assumptions in the three address code:

- All variables have the “integer” type.
- The programs have only global variables. Even the temporaries can be allocated globally.
- Function calls have no arguments and return only a single value (using a designated register).
- No floating point operations are present.