PPL Assignment-1

A-1

Questions
---

1) How do the declarations look in the GIMPLE? Specifically, what happens to multiple variables declared and initialized on the same line? What happens to global declaration [int Z]?

Ans: In Gimple variable each variable is declared on a separate line whereas in C the variable are initialized and declared on a single line. Gimple can handle only one operation at a time. So arithmetic expressions are broken down into small and simple operations consisting of only one operator. The result of these simple operations are stored in temporary variables. Unlike C, Global variables are not declared and initialized in Gimple…They are directly used in the program from memory.

2) How are complex expressions such as [c = a * b + 25] broken down? Can you determine a rule as to when temporary variables will be introduced?

Ans Gimple can handle only one operation at a time(consisting of a single operator).So Complex arithmetic expressions consisting of multiple operations are broken down into simple expressions denoting a single operation. Various temporary variables are created and used for storing the results of these simple operations except for the final result.This rule applies to any arithmetic expression composed of multiple operations.

3) How are floats/doubles represented in GIMPLE? What happens when a float/double is assigned to an integer in [Z = r]?

Ans In Gimple ,values of type float and double are converted and stored in the form of its scientific notation. When a float value is attempted to be assigned to an integer variable,the float value is typecasted to an int value and this new value is stored in a temporary variable. The value of the temporary variable is then assigned to the integer variable.

4) How are reads/writes to the global variable Z performed? Why is there a temporary introduced for the statement [Z = Z + 1] but not for the statements [q = Z] or [Z = p]?

Ans In Gimple, the global variables are not declared and initialized. They are read from the main memory and written into the program's memory from where they are directly used in our program. Gimple can handle only one operation at a time .So in the expression Z=Z+1,two operations are involved. One is to increment the value of Z by 1 and the second is to reassign the updated value of Z to the variable Z itself. So temporary variables are used.

A-2 Conditional Jumps and Control Flow Graphs

Compile the following program (file A-2.c) and observe the GIMPLE dumps.

Source file : A-2.c
Compilation : gcc-4.7.2 -c -fdump-tree-gimple A-2.c
View result : vi -O A-2.c A-2.c.*.gimple
Clear dumps : rm -f A-2.c.* A-2.o

Program

```
int Z;

void f()
{
        int a, b, c;
        a = Z;
        if (a < 10) {
                b = 5;
                c = 17;
        } else {
                b = 6;
                c = 20;
                if (a == 0) {
                        c = 0;
                }
        }
        Z = b * 10 + c;
}
```

Questions
---

1 How have if-blocks been translated into GIMPLE statements? Can you differentiate between conditional and unconditional gotos?

A.In Gimple depending on the truth value of the test expression the if else statements of C are divided into if else blocks with appropriate goto paths specified. Conditional Gotos are associated with conditional if-else statements which means the control will jump to the part of the code specified under goto if and only if a particular condition is true. Unconditional Gotos will be carried out anyway irrespective of any condition.

2 Repeat the above compilation, but now asking for another dump that of the pass "cfg". View this pass and say whether this representation makes it easier to read and understand GIMPLEs. Draw the control-flow graph of the program A-2.c on a piece of paper by looking at the CFG dump.

A.

Additional Problem

Try to figure out how GCC handles the ternary operator [a ? b : c] in GIMPLE.

Ans. Gimple ternary operations are handled the same way if else statements are handled.

A-3 Loops

Compile the following program (file A-3.c) and observe the dumps.

Source file : A-3.c
Compilation : gcc-4.7.2 -c -fdump-tree-cfg A-3.c
View result : vi -O A-3.c A-3.c.*.cfg
Clear dumps : rm -f A-3.c.* A-3.o

Program

```
int main()
{
        int sum = 0;
        int i, j;

        while(sum < 100) {
                sum = sum * 2;
        }

        for(i=0; i<25; i++) {
                for(j=0; j<50; j++) {
                        sum = sum + i*j;
                }
        }
}
```

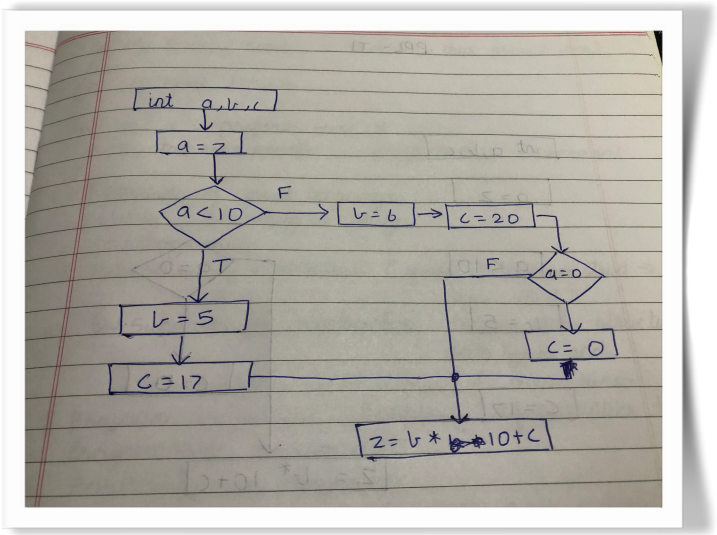
Questions
---

1 Identify the basic blocks that correspond to the while loop. How is the GIMPLE for a while loop different from simple if-else jumps?

Ans. 2,3,4 correspond to the while loop blocks. In the while loop blocks , the statements are executed in reverse order. The statements within the while loop that are to be executed when the loop test expression evaluates to true are enclosed in blocks which are specified before the block consisting of the while loop initialization. Whereas in if-else loops the statements to be executed when the condition evaluates to true are enclosed in blocks which are specified after the block consisting of the if else test expressions.

2 Identify the basic blocks for the nested for-loop. You should be able to identify intializers, conditions and increments for each loop as well as the shared inner body. Is there really anything special about a for-loop as compared to a while-loop?

Ans 6,7,8 correspond to the blocks for the nested for loop. There is no difference in the way for loops and while loops are written in Gimple.



A-4 Arrays and Pointers

In this program we observe how GCC treats arrays and pointers.

Compile the following program (file A-4.c) and observe the compiler output to answer the questions.

Source file : A-4.c
Compilation : gcc-4.7.2 -c -fdump-tree-cfg A-4.c
View result : vi -O A-4.c A-4.c.*.cfg
Clear dumps : rm -f A-4.c.* A-4.o

Program

```
int main()
{
   int a[3];
   int b[] = {1, 2, 3};
   int i, *p;
   for (i=0; i<3; i++) {
      a[i] = b[i];
   }
   p = a;
```

```
    *(p + 2) = 5;
}
```

Questions
---

1 How is array declaration and initialization handled?

Ans First the array name and size is declared, then the individual element of the array is initialised, then on separate lines different individual elements are declared. This happens because simple can handle only one operation at a time.

2 Why does the statement a[i] = b[i] get broken down to two steps with an intermediate temporary?

Ans

a[i]=b[i] gets broken down into two steps mainly because Gimple can handle only one task at time.The statement a[i]=b[i] takes place in two steps which are specified as follows:

(i) In the first step the memory Is allocated ant the value is assigned to the temporary variable

(ii)In the second step, the element of array a is assigned the value stored in the temporary variable.

3 Why are there CLOBBER statements at the end?

Ans CLOBBER statement is a way of telling the gcc compiler that a particular variable is no longer accessible at this point as it is undefined at this point.

A-5 Static Single Assignment

Now, we are going to take a look at the Static Single Assignment form that GCC uses later during optimization. In SSA form, each variable may be assigned at most once. Also, exactly one assignment of each variable dominates every use of the variable.

In order to implement SSA, local variables are suffixed with a unique number which represents the assignment. For example, two assignments to the variable `a' may look like `a_0' and `a_1'.

Compile the following program (file A-5.c) and observe the dumps.

Source file : A-5.c
Compilation : gcc-4.7.2 -c -fdump-tree-cfg -fdump-tree-ssa A-5.c
View result : vi -O A-5.c.*.cfg A-5.c.*.ssa
Clear dumps : rm -f A-5.c.* A-5.o

Program

```
int main()
{
   int a, b, c, d;

   d = 10;
   if (c > d) {
```

```
        a = 3;
        b = 2;
    } else {
        a = 2;
        b = 3;
    }

    c = a + b;
}
```

Questions
---

1 Do you notice that each usage of local variable as an operand uses a suffix that can be tracked to it's assignment? Why do some variables have a suffix `(D)'?

A.1)yes, some variable names consist of a numeric suffix without a (‘D’) , which indicates that they have been initialised. Whereas there are some variables where the numeric value is followed by a(‘D’) indicating that they have been declared but not initialized.

2 Find the point in the CFG in which the paths of the two assignments to variable `a' and `b' merge. You will notice that PHI statements have been introduced. Can you guess the syntax of the PHI statements?

A) In block 5(<bb 5>), the assignments to the two variables a,b merge.
The phi statements are included in block 5.
Syntax is <first_assignment(block in which it is assigned),second_assignment(block in which it is assigned)>

3 Can you think of an example in which a PHI node merges more than two versions of a variable?

A) Yes, in a if else-if ladder PHI nodes can merger more then two nodes to the same variable.

```
if(marks > 10){
        grade = 1
}
else if (grade> 20){
        grade = 2
}
else if (grade>40{
        grade=4
}
else if(grade > 50){
        grade = 5
}

grade = PHI<grade_3(2), grade_5(3), grade_2(4), grade_6(5)>
```

Good! You are half-way there. Now you know about how C is translated into GIMPLE. Let's see how good you are at understanding the C to GIMPLE conversion.

The following is a CFG dump of a somewhat meaningful little program. Can you predict the output of the program just by looking at the GIMPLE dump? Well, maybe or maybe not. But your task is to find the corresponding C program for which gcc-4.7.2 generated this GIMPLE dump.

A-6 GIMPLE CFG DUMP (file A-6-cfg-dump)

```
main ()
{
  int j;
  int i;
  int D.2168;

<bb 2>:
  i = 2;
  goto <bb 11>;

<bb 3>:
  j = 2;
  goto <bb 7>;

<bb 4>:
  D.2168 = i % j;
  if (D.2168 == 0)
    goto <bb 5>;
  else
    goto <bb 6>;

<bb 5>:
  goto <bb 8>;

<bb 6>:
  j = j + 1;

<bb 7>:
  if (j < i)
    goto <bb 4>;
  else
    goto <bb 8>;

<bb 8>:
  if (i == j)
    goto <bb 9>;
  else
    goto <bb 10>;

<bb 9>:
  printf ("%d \n", i);

<bb 10>:
  i = i + 1;
```

```
<bb 11>:
  if (i <= 100)
    goto <bb 3>;
  else
    goto <bb 12>;

<bb 12>:
  return;

}
```

Ans- OUTPUT:

```
int j;
int I;
int main(){
        for(i=2;i<=100;i++){
             for(j=2;j<i ;j++){
                 if(i%j==0){
                      if(i==j){
                           printf("%d\n",i);
                  }
          }
     }
}
```

```
}
```

B-1 Function Inlining

In this program we observe how the code of a function is placed in the position of its call if doing so does not increase the code size too much.

Source file : B-1.c
Compilation : gcc-4.7.2 -c -O2 -fdump-tree-all B-1.c
View result : vi -O B-1.c.*.ssa B-1.c.*.einline
Clear dumps : rm -f B-1.c.* B-1.o

Note: On some older versions of gcc the pass may be named "einline2".

Program

```
int AddTwo(int a)
{
   a = a + 2;
   return a;
}

int main()
{
   int x = 3;
   x = AddTwo(x);
   return x;
}
```

Questions
---

1 The body of AddTwo(x) seems to be inlined into the call of main rendering the function useless (nobody else is calling it). Then why is the definition of AddTwo(x) still there all the way till the last pass (B-1.c.*.optimised)?

A. The AddTwo(x) function is a function called from another file that is why it cannot be discarded, also it may not be private.

2 Is there any way to get rid of it?

A. To get rid of a function it should be declared with the keyword static so it remains private to the current working Directory, so then it will disappear if there is no call to it in the file.

B-3 Value Range Propagation

In Value Range Propagation, for each variable, a range (a pair of max-bound and min-bound) is maintained. Those conditional branches whose conditionals lie outside these bounds are eliminated.

Source file : B-3.c
Compilation : gcc-4.7.2 -c -O2 -fdump-tree-all B-3.c
View result : vi -O B-3.c.*.ssa B-3.c.*.vrp1
Clear dumps : rm -f B-3.c.* B-3.o

Program

```
#include <stdio.h>
int main ()
{
  int a, b;
  for (a=4 ; a<100; a++) {
     if (a < 4)
     b = b + 2;
     else
     b = b * 2;
  }
  printf ("%d%d", a, b);
}
```

Questions
---

1 In the file B-3.c, what values can variable `a' take? Will the `if' condition [a < 4] ever be true? Inspect the file B-3.c.*.vrp1 to analyse the dump.

A1. The variable 'a' can take values from 4-99 including 4 and 99. As the initial value will always minimum of 4, and the given condition is that 'a' should be less than 4 so the condition will never be true. Hence the loop will never be executed.

B-4 Common Subexpression Elimination

Here is a small program to observe the optimisation of Common Subexpression Elimination (CSE) or Full Redundancy Elimination (FRE). In the following program observe that the term "a + c" appears twice (albeit differently) and thus can be calculated once and the value is used next time when needed (the redundant calculation second time is eliminated, thus the term CSE/FRE).

You must demand -O2 for CSE/FRE.

Source file : B-4.c
Compilation : gcc-4.7.2 -c -O2 -fdump-tree-all B-4.c
View result : vi -O B-4.c.*.ssa B-4.c.*.fre1
Clear dumps : rm -f B-4.c.* B-4.o

Program

```
int main()
{
   int a, b, c;
   b = (a + c + b) * (c + a);
   return b;
}
```

Questions
---

1 How many times is the expression (a + c) computed in the original code (as seen in the SSA pass)? What about after the FRE pass?

A. According to the original code there are two computation, the FRE pass was able to reuse the temporary variable which is stores the result of the previous computation of(a+c). This helps us in saving the extra space of variable and also to save one line in the code.

2 Have there been any more optimisations after FRE? How will you check?

A.Check the file B-4.c.*.optimised. This is the last GIMPLE intra-procedural pass. If it is same as the output of FRE, then it means that no more optimisations have occurred.

3 Why did we have to put a "return b" in this assignment? What if "return b" is changed to "return 0"? Will it make any difference?

A. As we have used int main in the main function hence we need a return type, we have used ‘ return b’o make the computation of the variable ‘b’ meaningful. If the function is switched to ‘return 0’ then the GCC will consider that the variable b was a dead code and eliminate its dependencies one by one until the program reduces to a simple program like:

```
int main()
{
   return 0;
}
```