COMP26020 - Lab exercise for Part III (Compilers)

Register Allocation using Graph Colouring

Background

Computer programs, regardless of the programming language, often use many more variables than the number of variables that can fit in all CPU registers. When a program is compiled for execution on a given processor, the compiler needs to consider what variables will stay in registers and for how long. If we think that moving data from the memory takes several cycles, there is a performance benefit if the compiler can minimise such transfers. How to do this? By doing some 'clever' register allocation, for example, by making sure that the most frequently used variables are placed in registers.

To understand the problem, consider the following piece of code:

```
1. r1=x
2. r2=y
3. r3=r1*r2
4. r4=z
5. r5=r4+r2
6. r6=w
7. r7=r5+r6
8. r8=r7*r3
9. r9=r8+r1
```

In this piece of code, the programmer has used 9 variables. However, does this mean that 9 registers are needed? To find the answer, let us define the notion of a live range. For any given variable, there is a live range that starts from the point where a value is assigned to this variable and lasts until the last time this particular value is used. Note that if a new value is assigned to the same variable, a new live range starts. For example, a value for r2 is defined in instruction 2. The last time it is used is in instruction 5, hence, the live range is between 2 and 5. However, if instruction 4 was r2=z, the live range would be from 2 to 3 and another live range would start at instruction 4 and end at instruction 5.

To practice, you may want to find all live ranges of the code above. The answer is given: r1:[1,9], r2:[2,5], r3:[3,8], r4:[4,5], r5:[5,7], r6:[6,7], r7:[7,8], r8:[8,9], r9:[9,9].

Live ranges are important because they indicate how many values need to be live at any given instruction. For example, the live ranges above tell us that at instruction 6 four values need to be live. Clearly, the maximum number of values that need to be live at any instruction indicates how many registers we need to have so that all values (live ranges) can be placed in registers. However, what is most important, is that live ranges can guide register allocation: two live ranges that do not overlap or interfere can use the same register. For example, with the live ranges above, r2 and r6 can share the same register as the corresponding values are needed (or are 'live') at different parts of the code. Different algorithms have been developed to find how to allocate different live ranges to registers. This problem is known as register allocation. It is an NP-complete problem, which means that most of the different solutions proposed over the years are based on heuristics. For additional information you can find more in Chapter 13 of the 'Engineering a Compiler' recommended textbook:

https://www.sciencedirect.com/science/article/pii/B978012088478000013X

Among the different approaches, register allocation using graph colouring is a common approach. In register allocation using graph colouring, live ranges are used to create an interference graph. In this graph, every live range corresponds to a node. There is an edge between two nodes if the live ranges overlap. Then, register allocation becomes equivalent to the problem of graph colouring. This is a well-known graph theory problem where the aim is to colour all nodes of the graph so that two adjacent nodes do not share the same colour. Typically the goal is to find the smallest number of colours. Every colour corresponds to a register and the

colour of a node corresponds to the register that should be used for a particular live range. There are various algorithms to colour a graph. Here, we are going to focus on a simple algorithm, which is known as top-down colouring. The algorithm works as follows:

1. Assume an ordered list of colours (eg, red, black, blue, etc)
2. Assume an interference graph, where nodes are numbered: 1, 2, 3, …
3. Rank nodes (that is, live ranges) of the interference graph according to the number of neighbours in descending order. In case of a tie (that is, nodes with the same number of neighbours) the node with the lowest id takes priority.
4. Follow the ranking to assign colours from the list of colours. For each node, select the first colour from the list that is not used by the node's neighbours.
5. Keep following the ranking and repeating step 4 until all nodes are coloured.

Your task

Use a programming language of your choice to implement a program that:

- reads a file that lists an interference graph (input).
- writes a file that lists colours for every node of the graph (output).

The list of colours is given by the (upper-case) letters of the alphabet: A, B, C, …, Z. There is a total of 26 colours (or registers).

Input file specification:
A number of lines equal to the number of nodes of the interference graph. Every line contains the number of the node (in ascending order) and the numbers of all nodes with which there is an interference. Example:
1 2 3 4
2 1 4
3 1
4 1 2
This means that node 1 interferes with nodes 2, 3, 4. Node 2 interferes with nodes 1 (we knew this already) and 4. Node 3 interferes with node 1 and node 4 interferes with nodes 1,2.
You can assume that there are no more than 50 nodes.

Output file specification:
For every node (in ascending order), write node number and colour. For example:
1A
2B
3B
4C

Please make sure that you follow the specifications as most of the marking will take place automatically. Your executable should take two arguments, which indicate the name of the input file and the name of the output file. E.g.:

```
% myprogram.exe input.txt output.txt
```

You should be able to complete this task in 2 lab sessions. The deadline for submission is set after the 2nd lab session and before the 3rd lab session.

You should submit through gitlab. Your submission should include source file(s) and a readme file with instructions on how to compile and run your code.

Marking (out of 10) will take place according to the following scheme:

- 2 marks for code readability and sensible comments/explanation.
- 2 marks for implementing the example above correctly.
- 6 marks for finding the output of additional, automated tests correctly.