CSC 270H Assignment 2, Fall 2002
Storage allocation by graph colouring

Due at the end of Thursday October 24, 2002; no late assignments without written explanation.

Introduction

The history of high-level programming languages is one of increasingly sophisticated compilers, which allow the programmer to think in high-level terms and to worry less and less about such details of implementation as can be handled mechanically.

This assignment is concerned with one particular compiler optimization regarding space for variable storage. Consider a function which uses a variable named "departure_time" and another variable named "seating_count". If it so happens that the variable "departure_time" is only used in the first half of the function, and "seating_count" is only used in the second half of the function, they could actually use the same memory location without conflict.

We define the *live-range* of a variable as the set of program lines from the line that includes the first occurrence of the variable to the line that includes the last occurrence. If two variables have disjoint live-ranges, they can use the same computer storage (memory).

Traditionally, a programmer who notices this disjointness would re-use the same variable name for the two purposes to save memory space. Since the concepts underlying these two variables are unrelated, a suitable variable name is probably impossible; in practice, a comment probably explains the strange use of "departure_time" to indicate the seat capacity...

Asking the programmer to notice this situation and to perform this optimization by hand is obviously quite error-prone, and also is asking the programmer to perform by hand precisely the kind of low-level analysis which a high-level programming language is supposed to free us from. Furthermore, if at a future time the program is reorganized such that the live-ranges of these two variables are no longer disjoint, a nasty bug will be introduced if the programmer doesn't notice and separate the variables. However, if the storage-sharing is done automatically by the compiler, this problem does not arise.

In this assignment we will ignore non-sequential control flow (loops and selection (ifs)) so as to simplify the detection of the live-ranges for the individual variables. We will focus on the graph colouring algorithm used to allocate a minimal number of storage units, given the live-range information of a group of variables.

The graph colouring

This problem can be solved by graph-colouring because it has a straightforward translation to a graph-colouring problem. A "colouring" of a graph is an assignment of "colours" (usually small integers) to vertices such that every vertex is assigned a colour, and adjacent vertices always have different colours. Usually we are interested in using a minimal number of colours, or at least a small number.

The high-level language's idea of variables corresponds to vertices of a graph. A pair of variables' having an intersection in their live-ranges corresponds to an edge. (That is, variables with overlapping live-ranges are adjacent, i.e. there is an edge between the two variables' vertices, whereas variables with disjoint live-ranges are not adjacent, i.e. there is no edge between their vertices.) Then, a correct graph colouring is a valid assignment of storage units, represented by the colours, to variables. A minimal colouring is an optimal assignment of storage units to variables. Since intersecting live-ranges are represented by edges, variables which cannot use the same storage will not be able to be given the same colour.

We have discussed, or will shortly be discussing, the basic two representations of a graph: as an adjacency matrix or as a linked data structure with pointers representing edges. We would like you to think about which representation would be more appropriate for this problem. Since we don't want you to go in the wrong direction here, we will be posting the answer to this question on the Q&A web page. But please think about it and arrive at your own conclusion first.

Colouring algorithm

Sort the list of vertices in decreasing order by degree. Assign colour 0 to the highest-degree vertex and to all verticies not adjacent to the first vertex or to any other vertex already assigned colour 0. Then assign colour 1 to the first vertex not already coloured and to all uncoloured vertices not adjacent to any other vertex with colour 1. Repeat this process until all the vertices are coloured.

This method does not guarantee an optimal solution (the least number of colours necessary to colour the graph). Finding an optimal solution is an NP-complete problem; that is, the problem is intractable for large numbers of adjacencies. However, better algorithms than the above algorithm do exist. Nevertheless, for the purpose of this assignment, and indeed often for the requirements of a typical compiler, it will suffice to use the above algorithm.

Completing the symbol table

Starter code is in the directory ~ajr/a2 on the CDF machines. This starter code includes a (very small) portion of a compiler.

You will not have to change token.c, which turns an input stream of characters into a stream of tokens. That is, it will return ";" as a token, but also "53" as a single token, having decoded it as the integer 53; it also processes strings, and programming language identifiers (used as variable names, function names, etc).

However, before you can use token.c, you have to complete symbol.c, which keeps track of these identifiers. Mostly, you have to write symbol_lookup(), which finds an existing symbol in a "hash" data structure, or creates it if it's not present. There are instructions written in English where the body of symbol_lookup() should go.

Test the tokenizer and your symbol_lookup() by compiling with `gcc -Wall -ansi -pedantic testtoken.c token.c symbol.c emalloc.c`. For example, if you give the compiled program the input "`gloop=floop+2;`", it should output:

```
token #0 is the symbol gloop
token #1 is the character '='
token #2 is the symbol floop
token #3 is the character '+'
token #4 is the int const 2
token #5 is the character ';'
```

What to do

The supplied main program reads the tokens in a program, discarding operators, integers, and string values, and keeping only variables. When get_next_token() returns a TOKEN_IDENT, the identifier has already been added into the symbol table. We keep track of the fact that it was seen in this statement number. A semicolon indicates the end of a statement, so we increment our statement counter.

1. Get the tokenizer and symbol table working as above. You can leave aside most of the incomplete functions in symbol.c for now; just complete symbol_lookup(). After testtoken is working, you might want to move testtoken.c to a different directory or delete it (you can always get it back from ~ajr/a2/testtoken.c) so that you can say "`gcc -Wall -ansi -pedantic *.c`" to compile the full program.

2. Add the necessary functions to symbol.c, including symbol_set_seen(), and complete graph.c (leaving aside colour_graph() for now). Note the allocation of the arrays in new_graph(). Note that is_adjacent() is intended to return a boolean value (1 or 0).

3. Use the data from the symbol table, along with the "symbol iterator" data type as described in English at the end of main.c, to make a graph (in main.c) containing all symbols as vertices. Use symbol_setindex() to associate the index in the graph with the symbol in the symbol table.

4. Use the list of variables and their first and last occurrence statement numbers to find overlapping live-ranges of variables. Use two symbol iterators to go through all pairs of variables, checking the boolean live_range_is_overlapping(), which you need to implement in symbol.c. The formula for determining whether or not two variables' live ranges overlap is surprisingly short: the only way that the live-ranges can fail to intersect is if one of the live-ranges ends before the other begins.

5. Implement graph_colour() to colour the graph using the above algorithm. If you sort the graph's vertices in place, you will mess up the adjacency information, so instead you should sort a freshly allocated list of indices.

What to hand in

The deadline is Thursday October 24 at midnight. Late submissions are not normally accepted, and always require a written explanation. If you do have an explanation, please do submit your assignment late and send me an explanation by e-mail.

You are to submit files `main.c`, `graph.c`, and `symbol.c`. Each of these files **must** begin with a prologue comment stating your name, student number, and TA (or tutorial time/place), as well as the purpose of the program and any overall notes. Make sure your files are in plain text by using the `cat` command on a CDF machine to display them on the console.

Do **not** submit the `.h` files, `testtoken.c`, `token.c`, or `emalloc.c`, as the graders will use the original versions.

Once you are satisfied with your files, you can submit them for grading as follows:

```
submit -c csc270h -a a2 main.c graph.c symbol.c
```

Do not submit your compiled program. You may change your files and resubmit them any time up to the due time (the latest copy overwrites any earlier ones). You can check that your assignment has been submitted with the command:

```
submit -l -c csc270h -a a2
```

To resubmit a file that has already been submitted, use `-f`:

```
submit -c csc270h -a a2 -f symbol.c
```

Hand in your own work. A mark of zero is better than an academic penalty for plagiarism.

Your files must be written in C (not C++) and must compile and run on the CDF Unix system using `gcc -Wall -ansi -pedantic`, with the original `.h` files. They must follow the original specifications in the .h files, e.g. they should compile with `testtoken.c` as in the "Completing the symbol table" section above, or with any replacement for any of the modules which follows the specifications.