# Compilers — Homework 5
# $P_0$ register allocator

Phase 1 due: Wednesday, 17 Oct 2012, 20:00
Phase 2 due: Wednesday, 24 Oct 2012, 20:00

For this project, work in groups of two or three people. Groups containing both Bachelor and Master's students are not allowed, except with the instructors permission. Every member of the group is responsible for understanding the project implementation.

In this project, you will extend the compiler from Homework 3 with register allocation. You are responsible for fixing any bugs in that compiler. The source language is the same as the language in the earlier assignment.

Create a Python script named `compile.py` that takes the name of a file as a command-line argument. The file should contain the text of a $P_0$ program. The script should compile that program into x86 assembly code, which it prints to the standard output (stdout).

## 1 Phase 1: liveness analysis for $P_0$

Your compiler should, as in the earlier assignment, flatten $P_0$ ASTs into a sequence of statements each of which perform only one operation. This sequence of statements should then be translated into *pseudo-assembly*, that is, 32-bit x86 assembly code extended with virtual registers. Every variable (including temporaries) in the flattened $P_0$ program should be translated into a virtual register. Real registers should be used for the frame and stack pointers as needed, for function return values, and for instructions like `imull` where operands or results are hard-coded to particular registers. For example, the following (flattened) $P_0$ code might be translated into the corresponding pseudo-assembly.

```
t1 = 3
t2 = input()
x = t1 + t2
print x
```

```
movl 3, t1
call _input
movl %eax, t2
movl t1, x
addl t2, x
movl x, (%esp)
call _print_int_nl
```

Observe that function arguments are copied from virtual registers to the stack, and that the return value of `input` is copied from `%eax` to a virtual register.

To perform liveness analysis, you will need a representation of the pseudo-assembly that you can traverse. I suggest a list of instruction objects (e.g., a `MOV` object, a `CALL` object, ...) which may include objects representing their operands (register, virtual registers, memory (stack) locations, immediate values, ...).

The prologue and epilogue will be generated after register allocation and should not be represented explicitly in the pseudo-assembly.

For each program point (i.e., before and after each instruction), compute the set of live variables, including both real and virtual registers. Be careful with calls and instructions like `imull`. Assume calls write to the *caller-saves registers* `%eax`, `%ecx`, and `%edx`. You can ignore `%ebp` and `%esp` in your liveness analysis since these registers are always live and should not be allocated to virtual registers.

Since $P_0$ program is "`main`", the program need not set the return value, so you need not make `%eax` live after the last instruction. If you want, you can (in Phase 2 below) assign 0 into `%eax` in the epilogue—having `main` return 0 is the conventional way for Unix processes to indicate that they terminated successfully.

Modify your `compile.py` program to take additional command-line arguments as follows:

- If the option `-psuedo` is given, rather than compiling to assembly code, your script should instead print the pseudo-assembly instruction sequence and then exit.
- If the option `-liveness` is given, your script should print pseudo-assembly instruction sequence plus the set of live variables in comments and then exit. For example, for the above code, you should print:

```
# live:
movl 3, t1
# live: t1
call _input
# live: t1 %eax
movl %eax, t2
# live: t2 t1
movl t1, x
# live: t2 x
addl t2, x
# live: x
movl x, (%esp)
# live:
call _print_int_nl
# live:
```

## 2 Phase 2: register allocation for $P_0$

In this phase, you will extend `compile.py` to output register-allocated x86 assembly code. The output should compile with `gcc` and be linked with `runtime.c` to produce an executable.

Using the results of the liveness analysis you implemented in Phase 1, generate an interference graph. You should define Python classes to represent the graph. It may help with debugging to add a command-line option that prints the interference graph in the `dot` format so you can visualize it (see `graphviz.org`).

Color the graph with the three caller-saves registers `%eax`, `%ecx`, and `%edx`. For simplicity, do not use the callee-saves registers. `%ebx`, `%esi`, and `%edi` (if you were to use them, you must also take care to save them on the stack in the prologue and restore them in the epilogue.

Use the simplification algorithm outlined in class or the greedy algorithm in the lecture notes, pre-coloring the nodes corresponding to real physical registers.

If you need to spill a variable, rewrite the instruction sequence (possibly introducing new temporaries) and restart register allocation. You do not need to implement move coalescing or any heuristics to choose the best variables to spill—concentrate on generating correct code.

Once the graph is colored, rewrite the instruction sequence using the assigned registers. The sequence should now represent legal assembly code.

Compute the space needed for the stack frame and add the prologue and epilogue and any other boilerplate needed (e.g., the section header). Finally, output x86 assembly code.

As with the earlier compiler, you should be able to compile the code into an executable as follows:

```
$ python compile.py test1.py > test1.s
$ gcc -m32 test1.s runtime.c -o test1
```

## 3 Testing

When developing your compiler, you should use test-driven development. Write test cases—that is, $P_0$ input files—that exercise each feature of your compiler. You can reuse test cases from the interpreter assignment. You are encouraged to share test cases with your colleagues. Writing thorough tests before you write the compiler itself will help ensure that you cover all the cases correctly. Be sure to include both legal programs and illegal programs.

## 4 Submission

When submitting the assignment, include the names of all members of the group in each file of your submission. Only one member of the group needs to submit the code. If more than one submits, the later

submission will be graded. Submit both your source code and test cases. Code that does not parse or that fails because it calls undefined functions or tries to read undefined variables will be given a 0.

Both your code and tests will be graded. Be sure they are readable and well-documented.