Assignment 5
Instruction Scheduling, Function Pointers, Exceptions

15-411: Compiler Design
Frank Pfenning
Robbie Harwood, Sri Raghavan, Max Serrano

Due Tuesday, November 12, 2013 (1:30pm)

Reminder: Assignments are individual assignments, not done in pairs. The work must be all your own.

You may hand in a handwritten solution or a printout of a typeset solution at the beginning of lecture on Tuesday, November 12. Please read the late policy for written assignments on the course web page. If you decide not to typeset your answers, make sure the text and pictures are legible and clear.

Problem 1: Instruction Scheduling (20 points)

One of the techniques used in processor architecture to improve performance is *pipelining*. A pipelined processor divides instruction execution into stages, allowing several instructions to be executed at the same time.

For this problem, we will assume that the processor begin executing one instruction per cycle, and each instruction takes three cycles to complete.[1] We will also assume that the processor reads any registers required by an instruction at the beginning of the first cycle, and that the results of an instruction are visible in registers at the end of the third cycle. If an instruction would be pipelined, but depends on the value of a register which is currently being computed, the processor will *stall* until the value is ready.

For example, suppose we want to run the following small assembly snippet:

```
r0 <- r0 * 2
r1 <- r1 + 4
r2 <- r1 / 3
```

The instructions passing through the processor's pipeline would look as follows:

[1] On modern processors, this is a gross oversimplification. Superscalar execution, multiple dispatch, and other architectural features make the situation quite complicated. Out-of-order execution even to some extend supersedes the sort of instruction scheduling discussed here. See for example the Intel optimization manual: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html

cycle	stage 1	stage 2	stage 3
0	`r0 <- r0 * 2`		
1	`r1 <- r1 + 4`	`r0 <- r0 * 2`	
2		`r1 <- r1 + 4`	`r0 <- r0 * 2`
3			`r1 <- r1 + 4`
4	`r2 <- r1 / 3`		
5		`r2 <- r1 / 3`	
6			`r2 <- r1 / 3`

Note that the processor could not begin executing the third instruction until the second instruction was complete. Often, a compiler can mitigate this delay by (carefully) changing the order of the instructions it emits, called *instruction scheduling*. In this case, swapping the first and second lines of the assembly allows the processor to execute the instructions in one fewer cycle without changing the result of the program:

cycle	stage 1	stage 2	stage 3
0	`r1 <- r1 + 4`		
1	`r0 <- r0 * 2`	`r1 <- r1 + 4`	
2		`r0 <- r0 * 2`	`r1 <- r1 + 4`
3	`r2 <- r1 / 3`		`r0 <- r0 * 2`
4		`r2 <- r1 / 3`	
5			`r2 <- r1 / 3`

We will be working with the following abstract assembly:

```
1:    r1 <- r0
2:    r2 <- r1 + 4
3:    r3 <- r2 - 5
4:    r4 <- 5
5:    r5 <- r3 * 2
6:    r6 <- 8
7:    r7 <- r6 / r1
8:    r6 <- 10
9:    r8 <- r6 % r1
```

(a) On our hypothetical processor, how many cycles does it take to execute the code as given? You should show *some* supporting work rather than simply giving a final answer, but it does not need to be in the table format we used above (though this may be helpful to you!).

(b) Write down all dependencies between instructions. (Arrows between lines of code is an acceptable format for your answer.)

(c) Give a version of the program which schedules instructions to reduce execution time, making sure not to change the result of the program. How many cycles does your version of the program take to run? Again, provide some support for this number.

(d) Pipelining speeds up execution by taking advantage of parallelism in the program. What is the theoretical limit of this speedup? To answer this, imagine that rather than having one of each pipeline stage, our processor can begin executing arbitrarily many instructions in the same cycle, and find the minimum number of cycles required to execute the program.[2]

Problem 2: Function Pointers (20 points)

Function pointers, or some variant of them, are a common language feature that is especially prevalent in C and C++. You might even like the slogan "*Function Pointers are Values*". You probably remember passing a function pointer as an argument to `signal()` from 15-213, or using them to for client callbacks in advanced data structures in 15-122. Suppose we wanted to give C0 programmers the ability to declare variables as function pointers, assign functions to them, pass them as parameters to functions, call them, and even return them. Together with the polymorphism afforded by `void*`, this allows us to implement generic data structures in a (hypothetical, at this point) C1 language.

We will use the standard C syntax to declare function pointers, but only use a subset of the C syntax when it comes to using function pointers.

In order to do so, we will need to add the `&` (address-of) operator to the language. In C1 it will only be allowed to obtain the address of a function. In order to call a function

[2] One tool designed for a very similar sort of problem is the Gantt Chart: http://en.wikipedia.org/wiki/Gantt_chart

pointer, it must first be dereferenced. Therefore, if f is a function pointer, $f()$ is illegal, but $(*f)()$ is allowed.

For example, the following code would return 11 from `main`.

```
void increment_pair(int* a, int* b) {
    *a = *a + 1;
    *b = *b + 1;
}

void (*get_function())(int* x, int* y) {
    return &increment_pair;
}

typedef void (*fptr)(int* x, int* y);

void call_on_pair(int* a, int* b, fptr f) {
    return (*f)(a, b);
}

int main() {
    int* a = alloc(int); int* b = alloc(int);
    *a = 0; *b = 5;
    call_on_pair(a, b, &increment_pair);
    return 5 + *b;
}
```

(a) Describe, on a high level, what changes you would need to make to the grammar to handle function pointers. What nonterminals would need to be changed?

(b) Describe any changes to your program checker, especially the type checker, that would be needed to handle function pointers.

(c) Describe any additions or changes to the static semantics rules that would be needed to handle function pointers.

(d) Describe how you would modify the internal representation to allow function pointers. Describe which variables are used and defined by the modified IR instructions.

(e) Describe how adding function pointers would impact register allocation and instruction selection.

Problem 3: Exceptions (20 points)

Exceptions are a common language feature not present in C or C0 (though you probably recognize from the language you are using for your compiler—SML, Haskell, OCaml, and Java all support exceptions). Suppose we want to extend C0 with exceptions, so that the programmer can declare new flavours of exception, raise exceptions, and handle them. Our version of exceptions will not be very powerful, as they will not carry any data with them (aside from what flavour of exception was raised).

The semantics of our exceptions will be that when an exception is raised, it is handled by the nearest enclosing `try ... handle (...) ...` block. The exception is assigned to the identifier "argument" of the handle clause, which is available to the body of the handle clause. The syntax is as in the example on the next page.

(a) Describe any additions or changes to the static semantics that would be needed for exceptions.

(b) Describe how you would compile exceptions. Be sure you discuss all of the operations that need to be supported, and the changes to the runtime that may be required.

(c) In Assignment 3 we discussed optimizing tail calls. How would this optimization interact with your implementation of exceptions, if at all?

```
exception Small;
exception Big;

int hailstone(int x) {
    // check argument
    if (x < 1) {
        raise Small;
    }
    if (x > 9000) {
        // IT'S---too big, give up
        raise Big;
    }

    // recursive computation
    if (x == 1) {
        return 0;
    }
    if (x % 2 == 0) {
        return 1 + hailstone(x / 2);
    }
    return 1 + hailstone(3 * x + 1);
}

int main()
{
    int i = readint();

    while (true) {
        try {
            return hailstone(i);
        } handle (e) {
            if (e == Small) {
                // whoops, bad argument. try a bigger problem instead.
                i += 5;
            }
            if (e == Big) {
                // ooof. try a smaller problem instead.
                i /= 2;
            }
        }
    }

    return 0;
}
```