

CS3723 HOMEWORK #7 SOLUTIONS

CONTROL (Chapter 8)

1. What facilities for control flow are provided in hardware?

unconditional and conditional branches

2. (a) What is spaghetti code?

Code that makes use of lots of `gotos` that cause the control flow to jump around in unpredictable ways without an overall pattern.

- (b) Why is spaghetti code undesirable?

Spaghetti code is often difficult to maintain, since it is impossible to reason about small pieces of code in isolation, because you never know what code will be the destination of a jump and from where the jump will come.

3. Rewrite the following structured C program using `if`'s `goto`'s and labels instead of `while` and blocks (i.e., `{}`):

```
int i = 0;
while (i < 10) {
    i++;
}
```

```
L0: int i = 0;
L1: if (i >= 10) goto L4;
L2: i++;
L3: goto L1;
L4: ...
```

4. (a) What are exceptions?

A programming language construct for immediately exiting a nested group of currently active blocks, control structures, and/or functions and continue at a dynamically selected exception handler.

- (b) What is the original purpose for which exceptions were invented?

'Exceptional' situations (i.e., handling errors and other unusual circumstances)

- (c) What is an alternative use of exceptions?

As an optimization: avoiding unnecessary computation when the result is known earlier than it would be through the 'normal' path.

5. List the three effects of raising an exception.

(a) jump out of a block or function

(b) pass data as part of the jump

(c) return to a program point that was set up to continue the computation

6_α. List the two kinds of constructs provided by all exception mechanisms.

(1) a construct for raising exceptions (e.g., throw in Java), and (2) a construct for designating code to recover from the exception (e.g., catch in Java).

6_β. (a) Do exceptions have dynamic or static scoping?

dynamic

(b) Give and explain an example that illustrates the type of scoping of exceptions in ML, Java, or C++.

```
exception E;  
val f = ((fn x => 3 + x + (raise E) ) handle E => (fn y => y + 2));  
val z = (f(2) handle E => 42);
```

This example has two exception handlers for the exception E. One surrounds the definition of an anonymous function (to which the name f is later bound), and the other surrounds the call to f(2).

When f(2) is called in the definition of z, an exception occurs. If exceptions were statically scoped, the exception would be handled by the handler in the definition of f, which lexically surrounds the 'raise'. But, in fact, exceptions are dynamically scoped, so that handler would only get used if an exception were to occur during the definition of f (which will never happen). Instead, dynamic scoping binds the exception handler that surrounds the call f(2).

(c) Why is it desirable for exceptions to have this type of scoping?

This allows the flexibility for different callers to hook in different exception handlers and customize the error handling behavior.

7. Consider the ML program:

```
exception Signal of int;  
fun f(x) = if x=0 then raise Signal(0)  
          else if x=1 then raise Signal(1)  
          else if x=10 then raise Signal(x-8)  
          else (x-2) mod 4;  
f(10) handle Signal(0) => 0  
         | Signal(1) => 1  
         | Signal(x) => x+8;
```

(a) What is the type of f? *int->int*

(b) What is the type of f(10)? *int*

(c) What would be the type of raise Signal(0) in isolation? *'a*

(d) What is the type of raise Signal(0) in the context of the definition of f? *int*

(e) What is the type of x+8? *int*

(f) Is it required that the types of $f(10)$, 0 , 1 , and $x+8$ all match? Why or Why not?

Yes, because, the whole $f(10)$ handle $Signal(0) \Rightarrow 0 \mid Signal(1) \Rightarrow 1 \mid Signal(x) \Rightarrow x+8$ is one expression, so it has to have one unified type. But the value of the expression may be $f(10)$, 0 , 1 , or $x+8$ depending on whether or what kind of exception occurs during the execution of $f(10)$.

8. Do Exercise 8.1 in Mitchell.

- (a) *0. Twice is a higher-order function that applies the function that is passed to it (in this case, a predecessor function, $pred$) twice. The first time, $pred(1)$ is called, and it returns 0 . Then $pred(0)$ is called, and it raises $Excp(0)$. This exception is caught by $twice$ and 0 is returned.*
- (b) *1. $twice$ calls $dumb(1)$, which raises $Excp(1)$. And this is caught by $twice$ and returned.*
- (c) *1. $twice$ calls $smart(0)$, which calls $pred(0)$, which raises $Excp(0)$. This exception is caught by the $smart(0)$, which returns 1 . $twice$ then calls $smart(1)$, which calls $pred(1)$, which returns 0 . $smart(1)$ adds 1 to the 0 and returns 1 , which is then returned by $twice$.*

9. Do Exercise 8.2 in Mitchell.

Because ML evaluates operators such as $::$ left-to-right so the $hd(nil)$ will always be called before the $tl(nil)$, and so there will only ever be a Hd exception.

10. Do Exercise 8.4 in Mitchell. *call $f(11)$*

*call $f(9)$
call $f(7)$
call $f(5)$
call $f(3)$
call $f(1)$
raise Odd
pop AR for $f(1)$
pop AR for $f(3)$
handle Odd in $f(5)$
return -5 from $f(5)$
return -5 from $f(7)$
return -5 from $f(9)$
return -5 from $f(11)$*