**Aleksandr Itskovich (ai1221)**

1.

**Output:**

(A1)

(B1)

(A2) r=10001

(B2) r=20001

(A3) r=10002

(B3) r=20002

(A4) r=10003

**Explanation:**

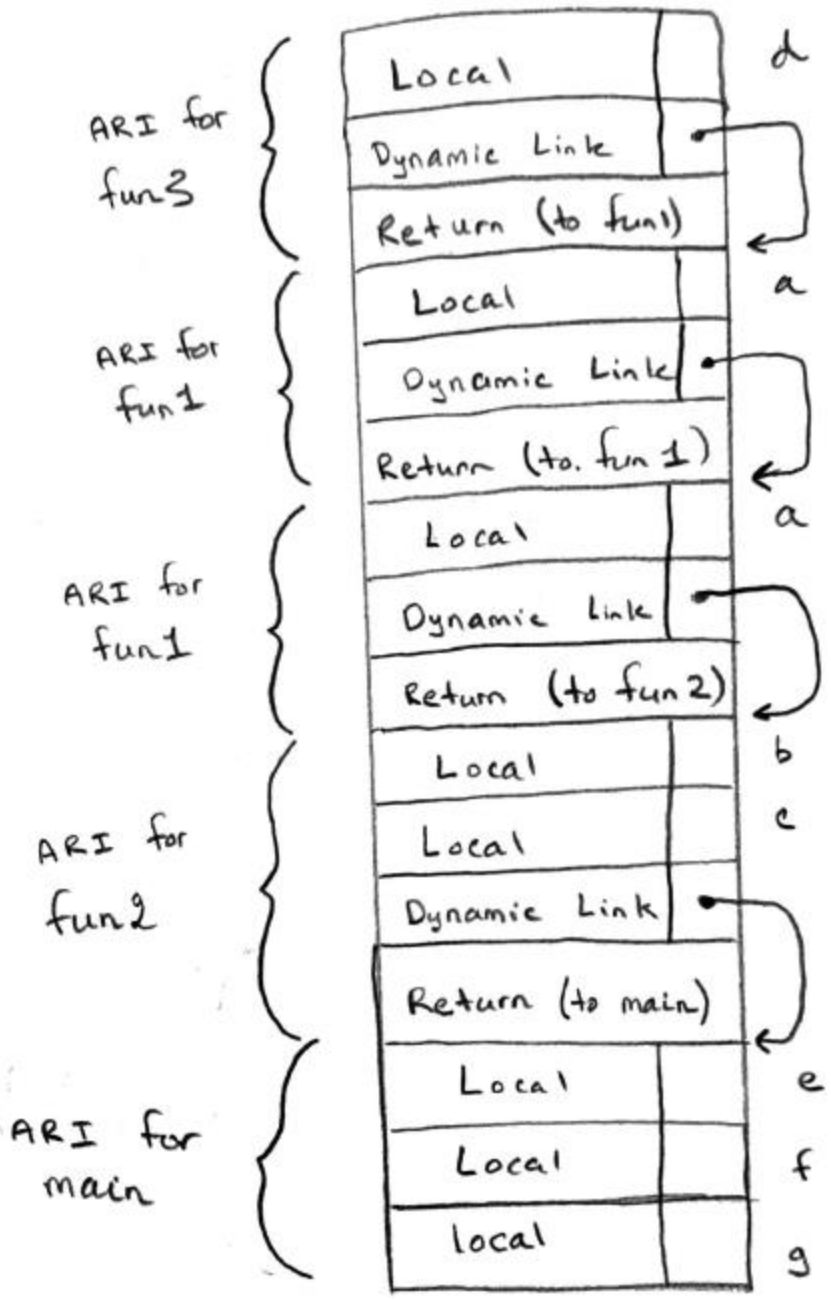
The program begins on line (36) with a call to routineA(). routineA() jumps to line (9) where it declares a local int variable r, then proceeds to print (A1). Next, on line (11) it calls setjmp with bufferA. The setjmp() function saves the current environment into a variable for later use by the function longjmp(). If setjmp() returns directly, it returns zero, but if it returns from a longjmp() function call then it returns the value passed to longjmp() as the second argument (the status). On line (11) we return 0 and save it in r. On line (12) because r is zero we call routineB().

routine() begins on line (24). Like routineA() we declare a local int variable r and then prints (B1) on line (25). Next, on line 26 it calls setjmp() with bufferB which returns zero (the behavior of setjmp() is described above). Because r is equal to 0 on line (27), the program calls longjmp with bufferA and a status of 10001. This takes us back to line (11) in routineA(). Now the local variable r is equal to 10001. Because of this, we pass over line (12) and print "(A2) r= 10001" on line (13). We set a new environment context on line (14), save zero in r and then call longjmp on bufferB and status 20001 on line (15). This takes us to line (26) in routineB(). The value of local r is now 20001. Because of this, we pass over line (27) and print "(B2) r=20001" one line (28). Next we save the environment context in routineB again on line (29). Then, because r is not 0, we call longjmp on bufferA and status 10002 on line (30). This takes us to line (14) in routineA because that is where the last environment context was saved. r is now 10002. We pass over line (15) and then print "(A3) r=10002" on line 16. We save the environment again on line (17), and call longjmp on bufferB with status 20002 on line (19). This takes us to line (29) in routineB(). r is now equal to 20002. We pass over line (30) and print "(B3) r = 20002" on line (31). We save the environment context again on line (32) and finally, because r is not zero, call longjmp with bufferA and status 10003 at line (33). This takes us to line (17) in routineA(). r is equal to 10003. We pass over line (18) and then print "(A4) r=10003" on line (20). This takes us back to main and the program ends.

Here are the line numbers for reference:

```c
#include <stdio.h>
#include <setjmp.h>


jmp_buf bufferA, bufferB;

void routineB();

void routineA() {
    int r;
    printf("(A1)\n");
    r = setjmp(bufferA);
    if (r == 0) routineB();
    printf("(A2) r=%d\n",r);
    r = setjmp(bufferA);
    if (r == 0) longjmp(bufferB, 20001);
    printf("(A3) r=%d\n",r);
    r = setjmp(bufferA);
    if (r == 0) longjmp(bufferB, 20002);
    printf("(A4) r=%d\n",r);
}

void routineB() {
        int r;
        printf("(B1)\n");
        r = setjmp(bufferB);
        if (r == 0) longjmp(bufferA, 10001);
        printf("(B2) r=%d\n", r);
        r = setjmp(bufferB);
        if (r == 0) longjmp(bufferA, 10002);
        printf("(B3) r=%d\n", r);
        r = setjmp(bufferB);
        if (r == 0) longjmp(bufferA, 10003);
}

int main(int argc, char **argv) {
    routineA();
    return 0;
}
```

2. We are implementing a dynamic-scoped program using deep-access. Because of this, the stack does not have static links which “serve no purpose in a dynamic-scoped language” (Sebesta, Ch. 10, page 463). The program needs to search through all the activation record instances on the stack to find a reference to any variable (which helps explain why dynamically-scoped languages are often slower than statically-scoped languages).




3.

*Mother relationship:*

Mother(X, Y) :- parent(X, Y), female(X)

*Sister relationship:*

We define mother and father first:

Mother(X, Y) :- parent(X, Y), female(X)

Father(X, Y) :- parent(X, Y), male(X)

sister(X, Y) :- female(X),

mother(M, X),

mother(M, Y),

father(F, X),

father(F, Y),

X /= Y

4.

```
function balanceBrackets(input::String)
  brackets = []
  partners = Dict(']' => '[', ')' => '(', '}' => '{')

  for c in input
    if c == '[' || c == '{' || c == '(' push!(brackets, c) end
    if c == ']' || c == '}' || c == ')'
      previousC = isempty(brackets) ? ' ' : pop!(brackets)
      expectedC = partners[c]
      if (previousC != expectedC) return false end;
    end
  end

  return isempty(brackets);
end

println(balanceBrackets("hello world"))
println(balanceBrackets("()(){}[]"))
println(balanceBrackets("()([]"))
println(balanceBrackets("()()}[]"))
println(balanceBrackets("("))
println(balanceBrackets(")"))
println(balanceBrackets("([{}({}[[{(())}]])])"))
```

```
true
true
false
false
false
false
true
```

5.

*What is Aspect-Oriented Programming?*

Aspect-Oriented programming (AOP) is a set of patterns that tries to make it easier to augment programs with reusable lines of code that can be “spliced in” while a program is running. It is a way of adding functionality to existing programs without actually having to have the program explicitly modifying the code itself. This additional behavior is a added as an **advice** (or aspect). An **advice** is a kind of function that modifies another function when the latter is being run. Typically, an advice represents some code feature that is reusable across multiple places in the code base and is orthogonal to the actual business logic (kind of like a helper function, but even more general). Advices are added into a

program at specified **join points**. Join points are points in the execution of a program, typically that are reachable by more than one code path. Join points are matched using **pointcuts**, which are expressions that identity specific join points as locations into which an advice should be added.

*What problem is Aspect-Oriented Programming trying to address?*

In programming there is a principle known as the **separation of cross-cutting concerns** to increase modularity. Cross-cutting concerns are issues that affect unrelated parts of the codebase in the same way. Examples include policy management (e.g. the scope of a session), security (i.e. who gets access to what part of the code), transactionality (i.e. batch code side-effects) and logging. However, it is difficult to obtain crosscutting in OOP because by code is divided across different objects by design. This is where AOP comes in. In AOP, cross-cutting concerns are encapsulated into first class abstractions known as aspects and these aspects are spliced into the code at some point before actual execution to provide the same behavior (e.g. logging) anywhere it is needed. The point is that now there are two entities: aspects and main classes, entirely separating from the OOP code itself, and can be elegantly woven in wherever they are required.

*What is AspectJ and what are its main features?*

AspectJ is an external aspect-oriented programming library for Java. AspectJ allows Java programmers to define their own **aspect** classes, allowing a programmer to add functions or fields from within the aspect class. The programmer can then specify the join points at which they would want to modify the original code, as well as specify what code they want to run at the join point (the **advice**). These constructs are called **pointcuts** within AspectJ, and they essentially allow a programmer to specify at what point in the execution a particular advice should be added. An important concept within AspectJ is **compile-time weaving**. The idea is that when the source code for both aspects and the original Java code, the AspectJ compiler will add the aspect code into the original Java classes and produce normal Java files as output that can be fed into the JVM. There are also other weaving strategies, such as port-compile weaving and load-time weaving. Modern AspectJ also allows the use of **annotations** to declare aspects. This makes it more similar to Python's decorators. AspectJ was one of the first available aspect-oriented libraries for Java, but not there are several including Spring AOP.

*What are some examples of AspectJ?*

```
public class Bathtub {
    int gallonsWater = 30;

    public boolean drain(int numGallonsToDrain) {
        if (numGallonsToDrain < gallonsWater) {
            return false;
        }

        gallonsWater = gallonsWater - numGallonsToDrain;
        return true;
    }
}
```

Create an Aspect class to log bathtub information:

```
public aspect BathtubAspect {

    pointcut callDrain(int numGallonsToDrain, Bathtub tub) :
```

```
        call(boolean Bathtub.drain(int)) && args(numGallonsToDrain) && target(tub);

    before(int numGallonsToDrain, Bathtub tub) : callDrain(numGallonsToDrain, tub) {}

    after(int numGallonsToDrain, Bathtub tub) : callDrain(numGallonsToDrain, tub) {}

    boolean around(int numGallonsToDrain, Bathtub tub) :
        callDrain(numGallonsToDrain, tub) {
            return (tub.gallonsWater < numGallonsToDrain)
                 ? false
                 : proceed(numGallonsToDrain, tub);
        }
}
```

This code defines a main Bathtub class that has an amount of water associated with it as well as a drain method. In addition, there is an aspect class that defines a pointcut on the drain method and 3 advises associated with the pointcut. After being woven in, the aspect class will log information about the drain method whenever it is called.

*How is AspectJ similar to Python Decorators and Haskell Monads?*

Unlike Java, Python is able to work in an Aspect-Oriented paradigm natively. Specifically, Python has decorators which essentially pass one function into another function, and replace the original function with the result. It is a way of modifying function behavior by supplying additional behavior at runtime. This is, in fact, similar to how AspectJ functions. AspectJ similarly defines some behavior outside of the main function body, and is able to splice its behavior in via weaving.

We can take a look at a custom Python decorator to further display the similarities:

```
def time(func):
  def _time (*args, **kwargs):
    t0 = time.time()
    ret = func(*args, **kwargs)
    print "time=%lf" % (time.time() - t0)
    return ret
  return _time

@time
def main_function(self):
  doSomething()
```

In Python, as in AspectJ, a behavior is being defined separately from the target function in order to modify it (in this case, adding information about program timing). Specifically, the timing code logic is defined in the time function, and is then added as behavior to a different main_function as a decorator. It is apparent that the same decorator can be used with any function, regardless of content, solidifying its crosscutting applicability. This is the same concern that AspectJ addresses. Of course, with Python there is no need to specify a particular pointcut or manage weaving because the language supports decorators out of the box.

Within functional programming languages, the traditional approach to handle crosscutting concerns and side-effects (such as logging) has been through the use of monads. Monads also allow for layering of

code, system-wide use and are easily integrated. Monads essentially do two things: allow the introduction of stateful computations or exceptions into purely functional languages, and provide an abstraction over different kinds of functions. For example, a Maybe monad represents a function that may fail for produce a result. While Monads can offer this type of flexibility, they need to be used by the programmer explicitly in the code. This is a point of difference from AOP where a programmer does not have to explicitly know about how an aspect is modifying their code. For example, in AspectJ a pointcut is defined externally and then modifies any part of the program in which it is applicable. There is also know way to declare a Monad in the way that a Python decorator does it, or the way that AspectJ allows it (using annotations). However, it can also be argued that they are somewhat similar to annotation-based AOP in that they can explicitly modify generic types with a certain behavior, as the Maybe example highlights. Of course, there are not explicit targets or pointcuts that are used.