Written Assignment 5
Due ??

This assignment asks you to prepare written answers to questions on type checking. Each of the questions has a short answer. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work.

1. Show the full type derivation (as done in slide 49 in the lecture notes) for the following judgement:

$$O[\text{Bool}/\text{x}] \vdash \texttt{x <- (let x:Object <- x in x = x): Bool}$$

2. Suppose we extend the grammar for Cool with a "**void**" keyword

$$\begin{array}{rcl} \textit{expr} & ::= & \textbf{void} \\ & | & ... \end{array}$$

that is analogous to `null` in Java. (Currently objects are initialized to void if they have no other initializer specified, but there is no general-purpose **void** keyword.) We want to be able to use **void** whereever an object can be used, as in

```
let foo:Int <- if some_test
               then 5
               else void
               fi
in ...
```

Give a sound typing rule that we can add to the Cool specification to accomodate this new keyword.

3. Suppose we extend Cool with exceptions by adding two new constructs to the Cool language.

$$\begin{array}{rcl} \textit{expr} & ::= & \textbf{try}\ \textit{expr}\ \textbf{catch}\ ID => \textit{expr} \\ & | & \textbf{throw}\ \textit{expr} \\ & | & ... \end{array}$$

Here **try**, **catch** and **throw** are three new terminals. "**throw** *expr*" returns *expr* to the closest dynamically enclosing catch block. Note that since **throw** expression returns control to a different location, we do not really care about the context in which throw is used. For example, (**throw** $false$) + 2 is a valid Cool expression (However, note that (**throw** $false$) + (2 + $true$) is not a valid Cool expression). Following is an example that uses the try-catch and throw constructs.

```
try
   if some_test1 then throw 34
   else if some_test2 then throw ''undefined error''
   else do_something fi fi
catch x =>
  case x of
      x:Int => do_something1
      x:String => do_something2
  esac
```

The above program fragment executes "do_something1" (with x bound to the value 34) if "some_test1" evaluates to $true$. It executes "do_something2" (with x bound to the value "undefined error") if "some_test1" evaluates to $false$ but "some_test2" evaluates to $true$. It executes "do_something" if both "some_test1" and "some_test2" evaluate to $false$.

Give a set of new sound typing rules that we can add to the Cool specification to accomodate these two new constructs.

4. The Java programming language includes arrays. The Java language specification states that if s is an array of elements of class S, and t is an array of elements of class T, then the assignment $s = t$ is allowed as long as T is a subclass of S. This typing rule for array assignments turns out to be unsound. (Java works around the fact that this rule is not statically sound by inserting runtime checks to generate an exception if arrays are used unsafely. For this question, assume there are no special runtime checks.)

   Consider the following Java program, which type checks according to the preceeding rule:

```
class Mammal { String name; }

class Dog extends Mammal { void beginBarking() { ... } }

class Main {
    static public void main(String argv[]) {
        Dog x[] = new Dog[5];
        Mammal y[] = x;

        /* Insert code here */
    }
}
```

   Add code to the `main` method so that the resulting program is a valid Java program (i.e., it type checks statically and so it will compile), but the program could result in an operation being applied to an inappropriate type when executed. Include a brief explanation of how your program exhibits the problem.

Solutions to Written Assignment 5

1. Show the full type derivation (as done in slide 49 in the lecture notes) for the following judgement:

$$O[\text{Bool}/\text{x}] \vdash \text{x <- (let x:Object <- x in x = x)}$$

Solution: Let `Bo` stand for the type `Bool` and `Ob` stand for the type `Object`.

$$\dfrac{\dfrac{O[\text{Bo}/\text{x}](\text{x}) = \text{Bo}}{O[\text{Bo}/\text{x}] \vdash \text{x: Bo}} \quad \text{Bo} \leq \text{Bo} \quad \dfrac{\dfrac{O[\text{Bo}/\text{x}] \vdash \text{x: Bo} \quad \text{Bo} \leq \text{Ob}}{O[\text{Bo}/\text{x}] \vdash \text{x: Ob}} \quad \text{Ob} \leq \text{Ob} \quad \dfrac{\dfrac{O[\text{Bo}/\text{x}][\text{Ob}/\text{x}](\text{x}) = \text{Ob}}{O[\text{Bo}/\text{x}][\text{Ob}/\text{x}] \vdash \text{x: Ob}} \quad \dfrac{O[\text{Bo}/\text{x}][\text{Ob}/\text{x}](\text{x}) = \text{Ob}}{O[\text{Bo}/\text{x}][\text{Ob}/\text{x}] \vdash \text{x: Ob}}}{O[\text{Bo}/\text{x}][\text{Ob}/\text{x}] \vdash \text{x = x: Bo}}}{O[\text{Bo}/\text{x}] \vdash \text{let x:Ob <- x in x = x: Bo}}}{O[\text{Bo}/\text{x}] \vdash \text{x <- (let x:Ob <- x in x = x): Bo}}$$

2. Suppose we extend the grammar for cool with a ``**void**'' keyword

$$\begin{array}{rcl} \textit{expr} & ::= & \textbf{void} \\ & | & \ldots \end{array}$$

that is analogous to null in Java. (Currently objects are initialized to void if they have no other initializer specified, but there is no general-purpose **void** keyword.) We want to be able to use **void** whereever an object can be used, as in

```
let foo:Int <- if some_test
               then 5
               else void
               fi
in ...
```

Give a sound typing rule that we can add to the Cool specification to accomodate this new keyword.

Solution:

$$\dfrac{}{O \vdash \texttt{void}\text{: T}}$$

You could also do this by defining a new type ``Void'':

$$\dfrac{}{O \vdash \texttt{void}\text{: Void}}$$

and declaring that Void $\leq$ T for all T. Note that your subtype graph is now a DAG, not a tree.

3. Suppose we extend Cool with exceptions by adding two new constructs to the cool language.

$$\begin{array}{rcl} \textit{expr} & ::= & \textbf{try}\ \textit{expr}\ \textbf{catch}\ \textit{ID} \Rightarrow \textit{expr} \\ & | & \textbf{throw}\ \textit{expr} \\ & | & \ldots \end{array}$$

Here **try**, **catch** and **throw** are three new terminals. ``**throw**$expr$'' returns $expr$ to the closest dynamically enclosing catch block. Note that since **throw** expression returns control to a different location, we do not really care about the context in which throw is used. For example, (**throw** $false$) + 2 is a valid Cool expression (However, note that (**throw** $false$) + (2 + $true$) is not a valid Cool expression). Following is an example that uses the try-catch and throw constructs. It executes ``do_something1'' (with $x$ bound to the value 34) if ``some_test1'' evaluates to $true$.

```
try
 if some_test1 then throw 34
 else if some_test2 then throw ''undefined error''
 else do_something fi fi
catch x =>
  case x of
      x:Int => do_something1
      x:String => do_something2
  esac
```

It executes ''do_something2'' (with x bound to the value ''undefined error'') if ''some_test1'' evaluates to $false$ but ''some_test2'' evaluates to $true$. It executes ''do_something'' if both ''some_test1'' and ''some_test2'' evaluate to $false$.

Give a set of new sound typing rules that we can add to the Cool specification to accomodate these two new constructs.

Solution:

$$\frac{O \vdash \texttt{e}: \quad T_1}{O \vdash \texttt{throw e}: \quad T_2}$$

$$\frac{O \vdash e_1 : \quad T_1 \qquad O[Object/x] \vdash e_2 : \quad T_2}{O \vdash \textbf{try } e_1 \textbf{ catch } x => e_2\textbf{:} \quad T_1 \sqcup T_2}$$

4. The Java programming language includes arrays. The Java language specification states that if s is an array of elements of class S, and t is an array of elements of class T, then the assignment $s = t$ is allowed as long as T is a subclass of S. This typing rule for array assignments turns out to be unsound. (Java works around the fact that this rule is not statically sound by inserting runtime checks to generate an exception if arrays are used unsafely. For this question, assume there are no special runtime checks.)

   Consider the following Java program, which type checks according to the preceeding rule:

```
class Mammal { String name; }

class Dog extends Mammal { void beginBarking() { ... } }

class Main {
    static public void main(String argv[]) {
        Dog x[] = new Dog[5];
        Mammal y[] = x;

        /* Insert code here */
    }
}
```

   Add code to the main method so that the resulting program is a valid Java program (i.e., it type checks statically and so it will compile), but the program could result in an operation being applied to an inappropriate type when executed. Include a brief explanation of how your program exhibits the problem.

   Solution:

```
Dog x[] = new Dog[5];
Mammal y[] = x;

Mammal a_cat = new Mammal();
y[0] = a_cat;                  // ###
x[0].beginBarking();
```

The problem here is that arrays are not just lists of values -- they represent memory locations into which we can store new data.

Normally we say that A $\leq$ B if objects of type A can safely be used anywhere that objects of type B can be used. That's not quite true with Dog[] $\leq$ Mammal[], since a Mammal[] object can be safely used on the left-hand side of the assignment marked by ###, while a Dog[] object cannot. Java handles this by adding a runtime check to every array assignment that determines whether the right-hand side of the assignment matches the dynamic type of the array.

Written Assignment 6
Due ??

This assignment asks you to prepare written answers to questions on run-time environment and code generation. Each of the questions has a short answer. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work.

1. Suppose `f` is a function with a call to `g` somewhere in the body of `f`:

```
f(...) {
 ... g(...) ...
}
```

We say that this particular call to `g` is a *tail call* if the call is the last thing `f` does before returning. For example, consider the following functions for computing positive powers of 2:

```
f(x:Int, acc:Int) : Int { if x > 0 then f(x-1, acc*2) else acc fi };
g(x:Int) : Int { if x > 0 then 2*g(x-1) else 1 fi };
```

Here `f(x, 1)` $=$ `g(x)` $= 2^x$ for $x \geq 0$. The recursive call to `f` is a tail call, while the recursive call to `g` is not. A function in which all recursive calls are tail calls is called *tail recursive*.

(a) Here is a non-tail recursive function for computing factorial:

```
fact(n:Int) : Int { if n > 0 then n*fact(n-1) else 1 fi };
```

Write a tail recursive function `fact2` that computes the same result. (Hint: Your function will most likely need two arguments, or it may need to invoke a function of two arguments.)

(b) Recall from lecture that function calls are usually implemented using a stack of activation records. Trace through the execution of `fact` and `fact2` both computing 4!, writing out the stack of activation records at each step (i.e., draw the tree of activation records). (If you were unable to write a tail-recursive version of `fact`, show functions `f` and `g` from above computing 2^4.) Indicate the amount of computation done before, during, and after each record is created or destroyed. Is there any place where you can see potential for making the execution of the tail-recursive `fact2` more time- or space-efficient than `fact` (without changing `fact2`'s source code)? What could you do?

(c) Now consider the following pair of functions:

```
f(x:Int, acc:Int) : Int { if x > 0 then g(x-1, acc*2) else acc fi };
g(x:Int, acc:Int) : Int { if x > 0 then f(x-1, acc*5) else acc fi };
```

In this case, the calls to `g` and `f` are all tail calls but they are not immediately recursive. Can you extend you answer to part (b) so that a compiler can use only one or two activation records for a call to `f` or `g`? (Hint: Consider the case when the initial invocation of these functions is via a call to `f` and the case when the initial invocation is via a call to `g`.)

2. Consider the following MIPS assembly code program. Using the code generation rules from lecture, what source program produces this code?

```
f_entry:
        move  $fp $sp
        sw    $ra 0($sp)
        addiu $sp $sp -4
        lw    $a0 4($fp)
        sw    $a0 0($sp)
```

```
        addiu $sp $sp -4
        li    $a0 0
        lw    $t1 4($sp)
        addiu $sp $sp 4
        beq   $a0 $t1 true_branch
false_branch:
        lw    $a0 4($fp)
        sw    $a0 0($sp)
        addiu $sp $sp -4
        sw    $fp 0($sp)
        addiu $sp $sp -4
        lw    $a0 4($fp)
        sw    $a0 0($sp)
        addiu $sp $sp -4
        li    $a0 1
        lw    $t1 4($sp)
        sub   $a0 $t1 $a0
        addiu $sp $sp 4
        sw    $a0 0($sp)
        addiu $sp $sp -4
        jal   f_entry
        lw    $t1 4($sp)
        add   $a0 $a0 $t1
        addiu $sp $sp 4
        b     end_if
true_branch:
        li    $a0 0
end_if:
        lw    $ra 4($sp)
        addiu $sp $sp 12
        lw    $fp 0($sp)
        jr    $ra
```

3. Give a recursive definition of the cgen function (as given in slide 52 in the lecture notes) for the following new construct.

$$\texttt{for i} = \texttt{e}_1 \texttt{ to e}_2 \texttt{ by e}_3 \texttt{ do e}_4$$

Assume that the subexpressions e_1, e_2, e_3 and e_4 are integer-valued. A "for loop" expression is evaluated according to the following rules. The first three subexpressions are evaluated once at the start of the loop in the order e_1, e_2, and then e_3. The subexpression e_4 is evaluated once per iteration of the loop. The index variable i is initialized to the value of e_1. The loop bound is the value of e_2 and i is incremented by the value of e_3 after each iteration. The loop terminates before executing an iteration where the value of i is greater than the loop bound. The value returned by the "for loop" expression is the value of the expression e_4 in the last iteration. If the loop does not execute at all, then the value returned is the integer 0.

Following is a more formal semantics of the for expression in terms of the Cool expressions.

let t: Int ← e_1 in
let bound:Int ← e_2 in
let incr:Int ← e_3 in
let result:Int ← 0 in
let i:Int ← t in
  while ($i \leq$ bound) loop {

```
        result ← e4;
        i ← i + incr;
    } pool;
    result
```

Note that the expressions e_1, e_2 and e_3 are evaluated ONLY once before the start of the loop. Also note that any occurences of variable i in e_1, e_2 and e_3 refer to the value of i just before the for loop. Any occurrence of variable i in expression e_4 refers to the loop index variable i.

Solutions to Written Assignment 6

1. (a) Here is a tail-recursive function that computes factorial:

```
fact2(n:Int, acc:Int) : Int { if n > 0 then fact2(n-1, n*acc) else acc fi };
```

Here `fact2(n, 1)` $= n!$. Alternately, if you wanted `fact2` to have only one parameter, you could have named the above function `fact3` and defined `fact2(n:Int) : Int { fact3(n,1) };`

(b) If we trace out the execution of `fact` and `fact2` computing 4!, we see that they both require at most five activation records on the stack. Below we show the stack during the evaluation of the last recursive call:

AR for `fact(4)`
AR for `fact(3)`
AR for `fact(2)`
AR for `fact(1)`
AR for `fact(0)`

AR for `fact2(4, 1)`
AR for `fact2(3, 4)`
AR for `fact2(2, 12)`
AR for `fact2(1, 24)`
AR for `fact2(0, 24)`

For `fact`, before a new activation record is pushed onto the stack we do one subtraction (decrease `n` by one). Just before we remove an activation record from the stack (i.e., just before we return) we do one multiplication (multiply `n*fact(n-1)`). For `fact2`, we do the subtraction and multiplication before each new record is pushed onto the stack. When we remove an activation record from the stack all we do is take the result of the recursive call to `fact2` and return it. Thus the only part of the stack frame for `fact2` we are using after a recursive call is the return address.

We can implement `fact2` more efficiently by reusing the same activation record for a recursive call, rather than pushing a new record on the stack. To recursively call `fact2`, we compute new values for `n` and `a` in temporary space, then to do the function call we replace `n` and `a` on the stack with their new values and restart `fact2`.

On the MIPS architecture, there are at least two ways to re-enter `fact2`. One choice is to retrieve the return address from the stack, store it in `$ra`, pop everything except the parameters off the stack, and then unconditionally jump to `fact2`'s entry point (we don't want to clobber `$ra`).

Another choice is to notice that the correct frame pointer and return address are already on the stack, and so we pop everything up to the return address off the stack, and then unconditionally jump just past `fact2`'s entry point to skip the initial set-up code.

In either case, no new space is required. With our new implementation, `fact2(n)` runs in constant space for any `n`, whereas `fact(n)` requires $O(\texttt{n})$ space.

For mutually tail-recursive `f` and `g` we need to be a little more careful than in part (b). In this case, `f` and `g` have the same number of arguments, so we could compile `f` and `g` into a single block of code with two entry points f and g. Then a function `h` that calls either `f` or `g` pushes two arguments onto the stack, stores the return address in `$ra` (on MIPS) and then jumps to either f or g. `f` or `g` modifies the arguments appropriately and then jumps to the other function (using one of the two strategies described above), or returns `acc` in the base case.

In general, however, `f` and `g` might have different signatures (e.g., take different numbers of parameters) but still be mutually tail-recursive, so we need a more general strategy.

To implement a general tail call to some function `h` we need to replace the current activation record with a new activation record that is correctly layed out for `h`. As before we compute the arguments to `h` in temporary space. Then we retreive the return address from the stack (on the MIPS we store it in `$ra`). Finally we overwrite the current activation record with the new parameters, shifting the stack pointer just past the last new parameter, and jump unconditionally to `h`.

2. The program that generates this code is

```
def f(n) = if n=0 then 0 else n+f(n-1)
```

3.
```
cgen(for i = e1 to e2 by e3 do e4, nt) =
            cgen(e1, nt)
            sw $a0 -nt*4($fp)
            cgen(e2, nt+1)
            sw $a0 -(nt+1)*4($fp)
            cgen(e3, nt+2)
            sw $a0 -(nt+2)*4($fp)
            li $a0 0
            sw $a0 -(nt+3)*4($fp)
            ld $a0 -nt*4($fp)
      loop: ld $t1 -(nt+1)*4($fp)
            bg $a0 $t1 finish
            cgen(e4, nt+4)
            sw $a0 -(nt+3)*4($fp)
            ld $a0 -nt($fp)
            ld $t1 -(nt+2)*4($fp)
            add $a0 $a0 $t1
            sw $a0 -nt*4($fp)
            j loop
    finish: ld $a0 -(nt+3)*4($fp)
```

Note that we assume that `nt` starts at 1 (at the beginning of a method) and increases by 1 for each temporary allocated (at the recursive `cgen` calls). In lecture, we started at 4 and incremented by 4 for each temporary allocated, but we do not multiply by 4 when computing the address of each temporary.

Written Assignment 7
Due ??

This assignment asks you to prepare written answers to questions on object layout and operational semantics. Each of the questions has a short answer. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work.

1. Consider the following Cool classes:

```
class A {
    attr1 : Int;
    attr2 : Int;
    method1() : Object { ... };
    method2() : Object { ... };
};

class B inherits A {
    attr3 : Int;
    method1() : Object { ... };
    method3() : Object { ... };
};
```

   (a) Draw a diagram that illustrates the layout of objects of type `A` and `B`, including their dispatch tables.

   (b) Let `obj` be a variable whose static type is `A`. Assume that `obj` is stored in register `$a0`. Write MIPS code for the function invocation `obj.method2()`. You may use temporary registers such as `$t0` if you wish.

   (c) Explain what happens in part (b) if `obj` has dynamic type `B`.

2. Suppose you wish to add arrays to Cool using the following syntax:

`let a:T[`e_1`] in `e_2	Create an array a with size e_1 of T's, usable in e_2
`a[`e_1`] <- `e_2	Assign e_2 to element e_1 in a
`a[e]`	Get element e of a

   Write the operational semantics for these three syntactic constructs. You may find it helpful to think of an array of type $T[n]$ as an object with n attributes of type T.

3. The operational semantics for Cool's `while` expression show that result of evaluating such an expression is always `void`. (See page 28 of the Cool manual.)

   However, we could have used the following alternative semantics:

   - If the loop body executes at least once, the result of the `while` expression is the result from the last iteration of the loop body.
   - If the loop body never executes (i.e., the condition is false the first time it is evaluated), then the result of the `while` expression is `void`.

   For example, consider the following expression:

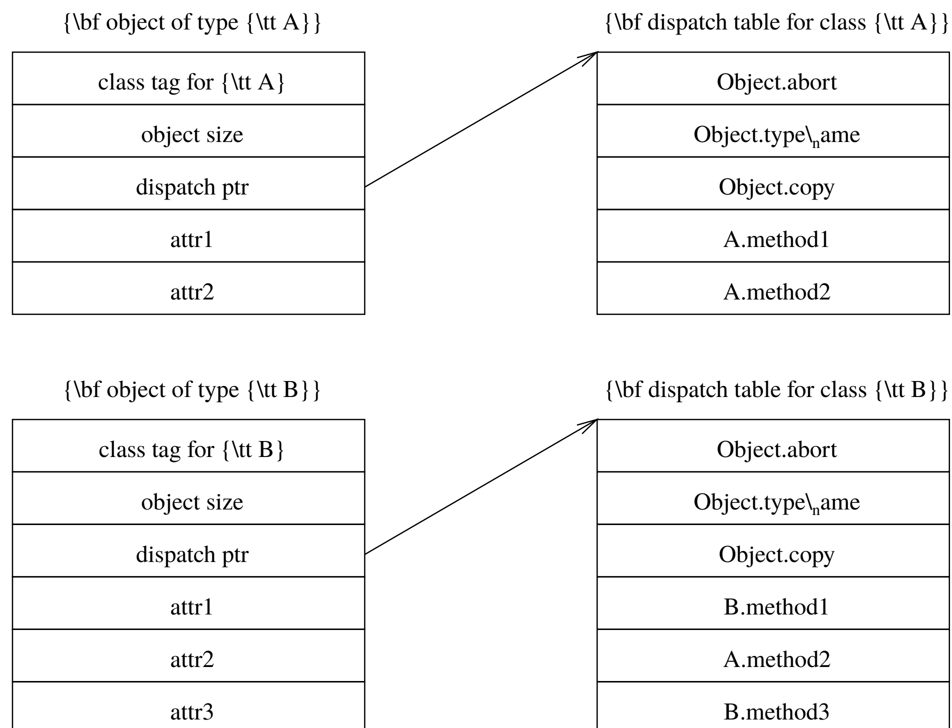
   ```
   while (x < 10) loop x <- x+1 pool
   ```

   The result of this expression would be 10 if `x` < 10 or `void` if `x` ≥ 10.

   Write new operational rules for the `while` construct that formalize these alternative semantics.

Solutions to Written Assignment 7

1. (a) The following diagram illustrates objects of type `A` and `B` along with their dispatch tables.



(b) Note that we assume the existence of a label called `dispatch_error` that handles the case where `$a0` is null.

```
beq    $a0 $zero dispatch_error       # check for null obj
lw     $t0 8($a0)                     # load dispatch pointer
lw     $t0 16($t0)                    # load method2 ptr from table
jalr   $t0                            # jump to method
```

(c) If `obj` has dynamic type `B`, then we correctly invoke `method2` on the object. All objects have a dispatch pointer at offset 8, so we correctly fetch the dispatch pointer. Furthermore, all classes that inherit from class `A` will have a pointer to the appropriate version of `method2` at offset 16 of their dispatch table. In this case, we will call the version of `method2` supplied by class `A`, since class `B` did not override it.

Note that it is legal to pass an object of type `B` as the `self` object, since the layout of `A` is a prefix of the layout of `B`.

2. These rules treat arrays as objects with n attributes, all of type T. For convenience, arrays are indexed from 1 to n, rather than 0 to $n - 1$.

$$\frac{\begin{array}{l} T = \begin{cases} X & \text{if } T_0 = \text{ SELF_TYPE and } so = X(\ldots) \\ T_0 & \text{otherwise} \end{cases} \\ so, S_1, E_1 \vdash e_1 : \text{Int}(n), S_2 \\ l_i = \text{newloc}(S_2) \text{ for } i = 0 \ldots n \text{ and each } l_i \text{ is distinct} \\ v_a = \text{array}(a_1 : l_1, \ldots, a_n : l_n) \\ S_3 = S_2[v_a/l_0, D_T/l_1, \ldots, D_T/l_n] \\ E_2 = E_1[l_0/a] \\ so, S_3, E_2 \vdash e_2 : v_2, S_4 \end{array}}{so, S_1, E_1 \vdash \text{let } a : T_0[e_1] \text{ in } e_2 : v_2, S_4} \quad \text{[Array-Let]}$$

D_T is just the default value for objects of type T (e.g., 0 for `Ints` and `void` for most objects).

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : \text{Int}(m), S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ E(a) = l_a \\ S_2(l_a) = v_a \\ v_a = \text{array}(a_1 : l_1, \ldots, a_n : l_n) \\ 1 \leq m \leq n \\ S4 = S3[v_2/l_m] \end{array}}{so, S_1, E \vdash a[e_1] \texttt{ <- } e_2 : v_2, S_4} \quad \text{[Array-Assign]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : \text{Int}(m), S_2 \\ E(a) = l_a \\ S_2(l_a) = v_a \\ v_a = \text{array}(a_1 : l_1, \ldots, a_n : l_n) \\ 1 \leq m \leq n \\ v = S_2(l_m) \end{array}}{so, S_1, E \vdash a[e] : v, S_2} \quad \text{[Array-Lookup]}$$

3. Here's one way to do it. This approach literally checks if this is the last time around the loop or not, and behaves accordingly. We also use the old false rule to handle loops where e_1 is never true.

$$\frac{so, S_1, E \vdash e_1 : Bool(false), S_2}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : void, S_2} \quad \text{[Loop-False]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : Bool(true), S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ so, S_3, E \vdash e_1 : Bool(false), S_4 \end{array}}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : v_2, S_4} \quad \text{[Loop-True-Last]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : Bool(true), S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ so, S_3, E \vdash e_1 : Bool(true), S_{peek} \\ so, S_3, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : v_3, S_4 \end{array}}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : v_3, S_4} \quad \text{[Loop-True-Not-Last]}$$

Note that S_{peek} is just thrown away; we're just using it to make sure that this isn't the last time around the loop.

On a real machine, we can't usually throw away a store like this; this solution is a good example of operationaly semantics being more powerful than we can easily implement. An alternate approach is to allocate a chunk of memory for the value of the loop expression, and write into it each time we traverse the loop.

Written Assignment 3
Due ??

This assignment asks you to prepare written answers to questions on type checking, run-time environments, and code generation. Each of the questions has a short answer. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work.

1. Consider the following class definitions.

```
class A {
  i : Int;
  o : Object;
  a : A <- new B;
  b : B <- new B;
  x : SELF_TYPE;
  f() : SELF_TYPE { x };
};
class B inherits A {
  g(b : Bool) : Object { (* EXPRESSION *) };
};
```

   Assume that the type checker implements the rules described in the lectures and in the Cool Reference Manual. For each of the following expressions, occurring in place of `(* EXPRESSION *)` in the body of the method `g`, show the static type inferred by the type checker for the expression. If the expression causes a type error, give a brief explanation of why the appropriate type checking rule for the expression cannot be applied.

```
1)  i + i
2)  x
3)  self = x
4)  self = i
5)  let x : B <- x in x
6)  case o of
        o : Int => b;
        o : Bool => o;
        o : Object => true;
    esac
7)  a.f().g(b)
8)  f()
```

2. Someone has proposed that Cool be extended to allow comparison, addition, and multiplication operations on `Bool` objects as well as on `Int` objects. The comparison, addition, and multiplication

operations are now defined for any combination of `Int` and `Bool` operands. An addition or multiplication operation involving an operand of type `Bool` produces a result of type `Int` (the `Bool` object is converted to 1 if it has the value `true`, and to 0 if it has the value `false`).

Write the additional type checking rules (as in the lecture and the Cool Reference Manual) for these operations on `Bool` objects.

3. (a) Your friend Damon feels constrained by the fact that every `while` expression in Cool evaluates to `void`. He would like to be able to write functions such as this:

```
f() : Bool {
  let x : Bool <- true in
    while x loop x <- false pool
};
```

Damon wants to change the semantics of the `while` expression so that the value of the expression is the value of the body on the last execution of the loop. He must define the value of a `while` expression in the case that the body of the loop is never evaluated, however.

Damon's first proposal is to define the value of the `while` expression to be `void` when the predicate of the loop is `false` initially and the body is never evaluated. He says that the type checker can now infer the static type of the `while` expression to be the static type of the body, because `void` is a member of every type. Give an example Cool program that shows how this change would cause a runtime type error (beyond the existing Cool runtime errors), and explain how the error occurs.

(b) After seeing your example, Damon comes up with a new suggestion. He proposes to eliminate the requirement that the predicate expression in a loop must be of static type `Bool`. Now, the predicate can have any static type, and a new method `is_true() : Bool` is added to the `Object` class.

The predicate is evaluated before each iteration of the loop. If the value of the predicate is `void`, the loop terminates. Otherwise, the method `is_true()` of the value of the predicate is invoked, and the loop terminates if the value returned by the method is `false`. If the value returned by the method is `true`, then the body of the loop is evaluated, and the process repeats. The value of the `while` expression is determined as follows:

- If the body of the loop is never evaluated, then the value of the `while` expression is the value of the predicate (from the first evaluation of the predicate).
- Otherwise, the value of the `while` expression is the value of the body on the last execution of the loop.

Can these modified `while` semantics be type checked statically to accept Damon's sample function above, while ensuring type safety (i.e., that no runtime type error will occur)? If so, write the most flexible type rule (the rule that accepts the most correct programs) for the modified `while` expression. If not, explain why not, and give an example Cool program that illustrates how this expression can introduce new runtime errors (beyond the existing Cool runtime errors).

(c) Damon wants to extend Cool by allowing method assignments. He would like to add a new assignment expression of the following form.

```
<exprB>.g <- <exprA>.f
```

Suppose that `<exprA>` evaluates to an object `a` of class `A`, and `<exprB>` evaluates to an object `b` of class `B`. Furthermore, `A` has a method named `f` and `B` has a method named `g`, and the two methods `f` and `g` have the same signature (the signature consists of the number of arguments, the types of the formal parameters, and the return type). The effect of the assignment would be to set the body of the method `g` of the object `b` to the body of the method `f` of the object `a`, so that subsequent invocations of the method `g` belonging to `b` would execute the body of the method `f` belonging to `a`. The value of the assignment expression would be `void`.

Damon says that, if `B` is a subclass of `A` (a descendant of `A` in the inheritance graph), then the inheritance rules of Cool guarantee that this operation is type safe.

Can Damon's method assignment expression be type checked statically to guarantee type safety? If so, write the most flexible type rule for the method assignment expression. If not, explain why not, and give an example Cool program that illustrates how this expression can introduce new runtime errors (beyond the existing Cool runtime errors).

4. Suppose `f` is a function with a call to `g` somewhere in the body of `f`.

```
f(...) {
 ... g(...) ...
}
```

We say that this particular call to `g` is a *tail call* if the call is the last thing `f` does before returning. For example, consider the following two functions for computing positive powers of 2.

```
f(x : Int, acc : Int) : Int { if (0 < x) then f(x - 1, acc * 2) else acc fi };
g(x : Int) : Int { if (0 < x) then (2 * g(x - 1)) else 1 fi };
```

Here `f(x, 1)` = `g(x)` = 2^x for $x \geq 0$. The recursive call to `f` is a tail call, while the recursive call to `g` is not. A function in which all recursive calls are tail calls is called *tail recursive.*

(a) Here is a non-tail recursive function for computing factorials.

```
fact(n : Int) : Int { if (0 < n) then (n * fact(n - 1)) else 1 fi };
```

Write a tail recursive function `fact2` that computes the same result. (Hint: Your function will most likely need two arguments, or it may need to invoke a function of two arguments.)

(b) Recall from lecture that function calls are usually implemented using a stack of activation records. Trace through the execution of `fact` and `fact2` as they compute 4!, showing the tree of activation records (each node of the tree shows the invocation of a function, and the arguments). How can a compiler make the execution of the tail recursive function `fact2` more efficient than that of `fact`? (Hint: Compare the stack space required for `fact(99)` with the stack space required for `fact2(99)`. Can `fact2` use fewer activation records?)

5. In some languages, a class can have multiple methods with the same name, as long as these methods differ in the number and/or types of formal parameters. This is referred to as method *overloading.*

Suppose we would like to add method overloading to Cool. Now, when generating code for a dispatch expression $e_0.f(e_1, \ldots, e_n)$, the compiler may need to choose the method to dispatch to (i.e., which slot in the dispatch table to jump to) amongst several valid possibilities. Let T_i be the static type of e_i for $i = 0, 1, \ldots, n$. Suppose that the compiler chooses a method f for the dispatch such that:

- T_0 has a method f with n formal parameters of types $P_1, \ldots, P_n$; and
- $T_i \leq P_i$ for $i = 1, \ldots, n$; and
- If T_0 has more than one method named f, then, for any other method named f with n formal parameters $Q_1, \ldots, Q_n$ satisfying $T_i \leq Q_i$ for $i = 1, \ldots, n$, it must be the case that $P_i \leq Q_i$ for $i = 1, \ldots, n$. In other words, $P_1, \ldots, P_n$ are the most specific parameter types for a method named f that could be invoked.

If a *unique* method exists under these rules, then the dispatch is accepted by the type checker. If more than one method satisfies these conditions, then the type checker signals a type error at compile time.

Method overriding occurs as described in the original Cool Reference Manual. Specifically, a method defined in a child class overrides any method with the identical signature in the parent class.

Consider the following Cool program:

```
class A inherits IO {
  f(a : Object, b : Object) : Object { out_string("1") };
  f(a : Object, b : Int) : Object { out_string("2") };
};
class B inherits A {
  f(a : Object, b : Object) : Object { out_string("3") };
  f(a : Int, b : Object) : Object { out_string("4") };
};
class Main {
  main() : Object {
    let a : A <- new B,
        b : B <- new B,
        x : Object <- new Object,
        y : Object <- 1,
        z : Int <- 2 in
      (* DISPATCH *)
  };
};
```

For each of the following dispatch expressions, give the output of the program when `(* DISPATCH *)` is replaced by the dispatch expression, or specify that a type error would occur.

```
a.f(x, x)        b.f(x, x)
a.f(x, y)        b.f(x, y)
a.f(x, z)        b.f(x, z)
a.f(y, x)        b.f(y, x)
a.f(y, y)        b.f(y, y)
a.f(y, z)        b.f(y, z)
a.f(z, x)        b.f(z, x)
a.f(z, y)        b.f(z, y)
a.f(z, z)        b.f(z, z)
```

Solutions to Written Assignment 3

1. Consider the following class definitions.

```
class A {
  i : Int;
  o : Object;
  a : A <- new B;
  b : B <- new B;
  x : SELF_TYPE;
  f() : SELF_TYPE { x };
};
class B inherits A {
  g(b : Bool) : Object { (* EXPRESSION *) };
};
```

Assume that the type checker implements the rules described in the lectures and in the Cool Reference Manual. For each of the following expressions, occurring in place of `(* EXPRESSION *)` in the body of the method `g`, show the static type inferred by the type checker for the expression. If the expression causes a type error, give a brief explanation of why the appropriate type checking rule for the expression cannot be applied.

1) `i + i`

`Int`

2) `x`

$\texttt{SELF_TYPE}_{\texttt{B}}$

3) `self = x`

`Bool`

4) `self = i`

Error: `Int` objects can only be compared with other `Int` objects

5) `let x :  B <- x in x`

`B`

```
6)  case o of
        o : Int => b;
        o : Bool => o;
        o : Object => true;
    esac
```

`Bool`

7) `a.f().g(b)`

Error: The class `A` does not have a method named `g`

8) `f()`

$\texttt{SELF_TYPE}_{\texttt{B}}$

2. Someone has proposed that Cool be extended to allow comparison, addition, and multiplication operations on `Bool` objects as well as on `Int` objects. The comparison, addition, and multiplication operations are now defined for any combination of `Int` and `Bool` operands. An addition or multiplication operation involving an operand of type `Bool` produces a result of type `Int` (the `Bool` object is converted to 1 if it has the value `true`, and to 0 if it has the value `false`).

   Write the additional type checking rules (as in the lecture and the Cool Reference Manual) for these operations on `Bool` objects.

   The original type checking rule for arithmetic operations remains the same for subtraction and division.

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : Int \\ O, M, C \vdash e_2 : Int \\ op \in \{-, /\} \end{array}}{O, M, C \vdash e_1 \; op \; e_2 : Int} \quad \text{[Arith]}$$

   A new rule is added for addition and multiplication.

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ T_1 \in \{Int, Bool\} \\ T_2 \in \{Int, Bool\} \\ op \in \{*, +\} \end{array}}{O, M, C \vdash e_1 \; op \; e_2 : Int} \quad \text{[Add-Mul]}$$

   The rule for non-equality comparisons must be extended to allow for `Bool` operands.

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ T_1 \in \{Int, Bool\} \\ T_2 \in \{Int, Bool\} \\ op \in \{<, \leq\} \end{array}}{O, M, C \vdash e_1 \; op \; e_2 : Bool} \quad \text{[Compare]}$$

   The rule for equality comparisons must be changed to allow for comparisons between `Int` and `Bool` operands.

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ T_1 = String \lor T_2 = String \Rightarrow T_1 = T_2 \\ T_1 \in \{Int, Bool\} \Rightarrow T_2 \in \{Int, Bool\} \\ T_2 \in \{Int, Bool\} \Rightarrow T_1 \in \{Int, Bool\} \end{array}}{O, M, C \vdash e_1 = e_2 : Bool} \quad \text{[Equal]}$$

3. (a) Your friend Damon feels constrained by the fact that every `while` expression in Cool evaluates to `void`. He would like to be able to write functions such as this:

```
f() : Bool {
  let x : Bool <- true in
    while x loop x <- false pool
};
```

Damon wants to change the semantics of the `while` expression so that the value of the expression is the value of the body on the last execution of the loop. He must define the value of a `while` expression in the case that the body of the loop is never evaluated, however.

Damon's first proposal is to define the value of the `while` expression to be `void` when the predicate of the loop is `false` initially and the body is never evaluated. He says that the type checker can now infer the static type of the `while` expression to be the static type of the body, because `void` is a member of every type. Give an example Cool program that shows how this change would cause a runtime type error (beyond the existing Cool runtime errors), and explain how the error occurs.

```
class Main {
  main() : Int {
    5 + while false loop 7 pool
  };
};
```

Under the proposed semantics, this program would be accepted by the type checker because the type checker infers the static type `Int` for the `while` expression. At runtime, however, the `while` expression evaluates to `void`, and so an integer is added to `void`. The result of this addition is undefined, as under the standard definition of Cool an expression whose static type is `Int` cannot take the value `void` at runtime. As such, a new runtime error is introduced into Cool.

(b) After seeing your example, Damon comes up with a new suggestion. He proposes to eliminate the requirement that the predicate expression in a loop must be of static type `Bool`. Now, the predicate can have any static type, and a new method `is_true() : Bool` is added to the `Object` class.

The predicate is evaluated before each iteration of the loop. If the value of the predicate is `void`, the loop terminates. Otherwise, the method `is_true()` of the value of the predicate is invoked, and the loop terminates if the value returned by the method is `false`. If the value returned by the method is `true`, then the body of the loop is evaluated, and the process repeats. The value of the `while` expression is determined as follows:

- If the body of the loop is never evaluated, then the value of the `while` expression is the value of the predicate (from the first evaluation of the predicate).
- Otherwise, the value of the `while` expression is the value of the body on the last execution of the loop.

Can these modified `while` semantics be type checked statically to accept Damon's sample function above, while ensuring type safety (i.e., that no runtime type error will occur)? If so, write the most flexible type rule (the rule that accepts the most correct programs) for the modified `while` expression. If not, explain why not, and give an example Cool program that illustrates how this expression can introduce new runtime errors (beyond the existing Cool runtime errors).

$$\frac{O, M, C \vdash e_1 : T_1 \quad O, M, C \vdash e_2 : T_2}{O, M, C \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : T_1 \sqcup T_2} \text{[Loop]}$$

Under this type rule, the basic types `Int`, `String`, and `Bool` will be inferred by the type checker for a `while` expression only if the type checker infers the same basic type for both the predicate and the body of the loop. The existing rules of Cool do not allow an expression with static type `Int`, `String`, or `Bool` to take the value `void` at runtime. As a result, a `while` expression for which the type checker infers one of the static types `Int`, `String`, or `Bool` will not evaluate to `void`. The only runtime errors caused by the `while` expression are the runtime errors due to `void` that already exist in Cool.

(c) Damon wants to extend Cool by allowing method assignments. He would like to add a new assignment expression of the following form.

```
<exprB>.g <- <exprA>.f
```

Suppose that `<exprA>` evaluates to an object `a` of class `A`, and `<exprB>` evaluates to an object `b` of class `B`. Furthermore, `A` has a method named `f` and `B` has a method named `g`, and the two methods `f` and `g` have the same signature (the signature consists of the number of arguments, the types of the formal parameters, and the return type). The effect of the assignment would be to set the body of the method `g` of the object `b` to the body of the method `f` of the object `a`, so that subsequent invocations of the method `g` belonging to `b` would execute the body of the method `f` belonging to `a`. The value of the assignment expression would be `void`.

Damon says that, if `B` is a subclass of `A` (a descendant of `A` in the inheritance graph), then the inheritance rules of Cool guarantee that this operation is type safe.

Can Damon's method assignment expression be type checked statically to guarantee type safety? If so, write the most flexible type rule for the method assignment expression. If not, explain why not, and give an example Cool program that illustrates how this expression can introduce new runtime errors (beyond the existing Cool runtime errors).

This method assignment expression cannot be type checked statically while ensuring type safety because the dynamic types of the objects involved in the assignment may be different from the static types. The type checker cannot guarantee at compile time that the dynamic type of the object to which the method is being assigned is a subclass of the dynamic type of the object that contains the method being assigned.

```
class A {
  x : Int;
  f(z : Int) : Int {
    z + x
  };
};
class B inherits A {
  y : Int;
  f(z : Int) : Int {
    z + x + y
  };
};
```

```
class Main {
  main() : Object {
    let a : A <- new A,
        b : A <- new B in {
      a.f <- b.f;
      a.f(1);
    }
  };
};
```

Any static type rule for the method assignment expression would have to allow the method assignment `a.f <- b.f` in this program, because the static types of `a` and `b` are the same. However, executing this program would lead to a runtime error because the object `a` does not have an attribute `y`.

4. Suppose `f` is a function with a call to `g` somewhere in the body of `f`.

```
f(...) {
 ... g(...) ...
}
```

We say that this particular call to `g` is a *tail call* if the call is the last thing `f` does before returning. For example, consider the following two functions for computing positive powers of 2.

```
f(x : Int, acc : Int) : Int { if (0 < x) then f(x - 1, acc * 2) else acc fi };
g(x : Int) : Int { if (0 < x) then (2 * g(x - 1)) else 1 fi };
```

Here `f(x, 1)` $=$ `g(x)` $= 2^x$ for $x \geq 0$. The recursive call to `f` is a tail call, while the recursive call to `g` is not. A function in which all recursive calls are tail calls is called *tail recursive*.

(a) Here is a non-tail recursive function for computing factorials.

```
fact(n : Int) : Int { if (0 < n) then (n * fact(n - 1)) else 1 fi };
```

Write a tail recursive function `fact2` that computes the same result. (Hint: Your function will most likely need two arguments, or it may need to invoke a function of two arguments.)

```
fact2(n : Int, acc : Int) : Int {
  if (n = 0)
  then acc
  else fact2(n - 1, acc * n)
  fi
};
```

(b) Recall from lecture that function calls are usually implemented using a stack of activation records. Trace through the execution of `fact` and `fact2` as they compute 4!, showing the tree of activation records (each node of the tree shows the invocation of a function, and the arguments). How can a compiler make the execution of the tail recursive function `fact2` more efficient than that of `fact`? (Hint: Compare the stack space required for `fact(99)` with the stack space required for `fact2(99)`. Can `fact2` use fewer activation records?)

The function invocations, with arguments, are as follows.

$$\texttt{fact}(4) \rightarrow \texttt{fact}(3) \rightarrow \texttt{fact}(2) \rightarrow \texttt{fact}(1) \rightarrow \texttt{fact}(0)$$
$$\texttt{fact2}(4,1) \rightarrow \texttt{fact2}(3,4) \rightarrow \texttt{fact2}(2,12) \rightarrow \texttt{fact2}(1,24) \rightarrow \texttt{fact2}(0,24)$$

A compiler can compile a tail recursive function such as `fact2` so that a new activation record is not needed for every recursive invocation of `fact2` in a computation. Suppose that the method `fact2` is invoked with the arguments 4 and 1 from a function `main`. The return address in the activation record for this invocation is an address R that specifies an instruction in the body of `main`.

Since `fact2` is tail recursive, after the recursive invocation of `fact2` with the arguments 3 and 4 has completed, no further computation must be done in the body of `fact2` to complete the invocation `fact2(4, 1)`. The return value of `fact2(3, 4)` is the return value of `fact2(4, 1)`. As a result, the compiler can replace the activation record for `fact2(4, 1)` with the activation record for `fact2(3, 4)`. The return address for `fact2(3, 4)` is the same as the return address for `fact2(4, 1)` (R), and the return value of `fact2(4, 1)` is now set during the execution of `fact2(3, 4)`. Figure 1 illustrates the stack space saved by this compiler optimization, showing the activation records during the invocation `fact2(3, 4)`.

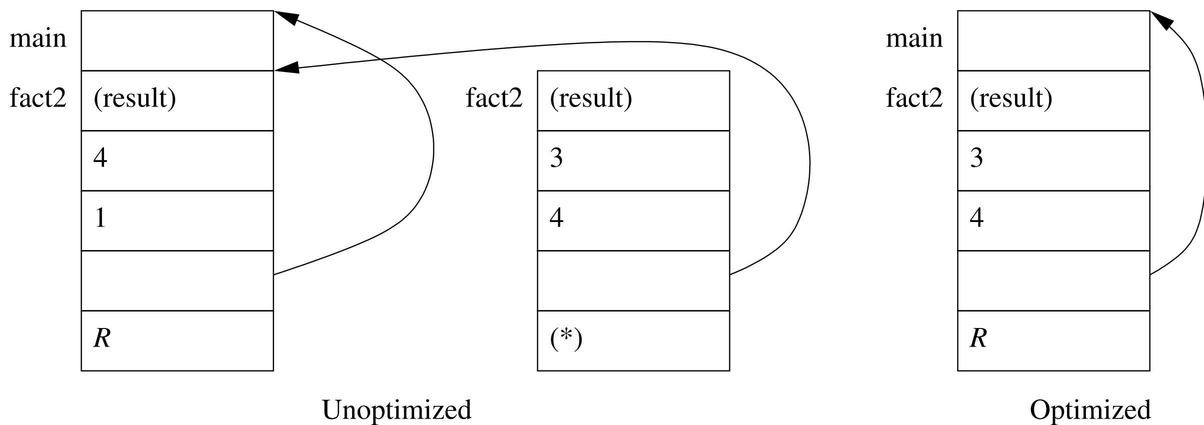


Figure 1: Activation records during the function invocation `fact2(3, 4)`.

5. In some languages, a class can have multiple methods with the same name, as long as these methods differ in the number and/or types of formal parameters. This is referred to as method *overloading*.

   Suppose we would like to add method overloading to Cool. Now, when generating code for a dispatch expression $e_0.f(e_1, \ldots, e_n)$, the compiler may need to choose the method to dispatch to (i.e., which slot in the dispatch table to jump to) amongst several valid possibilities. Let T_i be the static type of e_i for $i = 0, 1, \ldots, n$. Suppose that the compiler chooses a method f for the dispatch such that:

   - T_0 has a method f with n formal parameters of types $P_1, \ldots, P_n$; and
   - $T_i \leq P_i$ for $i = 1, \ldots, n$; and
   - If T_0 has more than one method named f, then, for any other method named f with n formal parameters $Q_1, \ldots, Q_n$ satisfying $T_i \leq Q_i$ for $i = 1, \ldots, n$, it must be the case that $P_i \leq Q_i$

for $i = 1, \ldots, n$. In other words, $P_1, \ldots, P_n$ are the most specific parameter types for a method named f that could be invoked.

If a *unique* method exists under these rules, then the dispatch is accepted by the type checker. If more than one method satisfies these conditions, then the type checker signals a type error at compile time.

Method overriding occurs as described in the original Cool Reference Manual. Specifically, a method defined in a child class overrides any method with the identical signature in the parent class.

Consider the following Cool program:

```
class A inherits IO {
  f(a : Object, b : Object) : Object { out_string("1") };   (* offset 0 *)
  f(a : Object, b : Int) : Object { out_string("2") };      (* offset 1 *)
};
class B inherits A {
  f(a : Object, b : Object) : Object { out_string("3") };   (* offset 0 *)
  f(a : Int, b : Object) : Object { out_string("4") };      (* offset 2 *)
};
class Main {
  main() : Object {
    let a : A <- new B,
        b : B <- new B,
        x : Object <- new Object,
        y : Object <- 1,
        z : Int <- 2 in
      (* DISPATCH *)
  };
};
```

For each of the following dispatch expressions, give the output of the program when `(* DISPATCH *)` is replaced by the dispatch expression, or specify that a type error would occur.

The compiler chooses an offset in the dispatch table for a method invocation at compile time, based on the static types of the arguments. When the invocation occurs at runtime, the method that executes is the method at that offset in the dispatch table. The following table shows the method offset chosen by the compiler (assuming the offsets are fixed according to the numbers in the comments in the program) and the output for each of the dispatch expressions.

Dispatch	Offset	Output
`a.f(x, x)`	0	`3`
`a.f(x, y)`	0	`3`
`a.f(x, z)`	1	`2`
`a.f(y, x)`	0	`3`
`a.f(y, y)`	0	`3`
`a.f(y, z)`	1	`2`
`a.f(z, x)`	0	`3`
`a.f(z, y)`	0	`3`
`a.f(z, z)`	1	`2`
`b.f(x, x)`	0	`3`
`b.f(x, y)`	0	`3`
`b.f(x, z)`	1	`2`
`b.f(y, x)`	0	`3`
`b.f(y, y)`	0	`3`
`b.f(y, z)`	1	`2`
`b.f(z, x)`	2	`4`
`b.f(z, y)`	2	`4`
`b.f(z, z)`	1 or 2	TYPE ERROR