Mandatory 1

- Any questions?

- Some hints:
  - No need to scan or parse. Can initiate with creation of objects. Eg. Something like:
    xxx.interpret([new Statement(new Expression), ….] ….);
  - It is OK to assume that programs are written in a relatively reasonable manner; you don't have to take every potential error into consideration

- Please include the example programs in your delivery
  - It is better to deliver a solution that almost works, than nothing at all

Problem 1

```
type S1 is struct {
      int y;
      int w;
};
type S2 is struct {
      int y;
      int w;
};
type S3 is struct {
      int y;
};
S3 f(S1 p) { ... };
...
S1 a, x;
S2 b;
S3 c;
int d;
```

```
a = b;        // (1)
x = a;        // (2)
c = f(b);     // (3)
d = f(a);     // (4)
```

Which of these four statements are correct under
a) name compatibility?
b) structural compatibility?

Problem 1

```
type S1 is struct {
      int y;
      int w;
};
type S2 is struct {
      int y;
      int w;
};
type S3 is struct {
      int y;
};
S3 f(S1 p) { ... };
...
S1 a, x;
S2 b;
S3 c;
int d;
```

```
a = b;      // (1)
x = a;      // (2)
c = f(b);   // (3)
d = f(a);   // (4)
```

Which of these four statements are correct under
a) name compatibility?
b) structural compatibility?

Name compatibility

a = b; is incorrect	Different type
x = a; is correct	
c = f(b); is incorrect	Ret diff type
d = f(a); is incorrect	Ret diff type

Problem 1

```
type S1 is struct {
       int y;
       int w;
};
type S2 is struct {
       int y;
       int w;
};
type S3 is struct {
       int y;
};
S3 f(S1 p) { ... };
...
S1 a, x;
S2 b;
S3 c;
int d;
```

```
a = b;        // (1)
x = a;        // (2)
c = f(b);     // (3)
d = f(a);     // (4)
```

Which of these four statements are correct under
a) name compatibility?
b) structural compatibility?

Structural compatibility

a = b; is correct

x = a; is correct

c = f(b); is correct

d = f(a); is incorrect Struct vs int

Problem 2

We have the following classes:

```
class Food { … }
class Cheese extends Food { … }
```

Assume that we have the following functions:

```
int f (Cheese c) { … }

int f' (Food f) { .. } // in this language, f' is just an ordinary name
```

someFood is a value of type Food, and someCheese is a value of type Cheese. Then we know that

f'(someCheese) can be substituted for f(someCheese)

that is, whenever we have a call 'f(someCheese)' we may just as well call f' with the same someCheese parameter without causing any static type errors: f' can be said to be a subtype of f.

Why cannot f(someFood) be substituted for f'(someFood)? That is why can not f said to be a subtype of f'? Give an example of class Cheese (that is a more elaborate Cheese than above) and a definition of f that will create a type error.

Problem 2

The answer is of course that f expects a more specific type than Food, namely Cheese. Cheese might have properties that Food does not.

For instance, imagine this definition of Cheese:

```
class Cheese extends Food { void melt() { … } }
```

and this definition of f:

```
int f (Cheese c) { … c.melt(); … }
```

Clearly, a call to f(someFood) would not work here.

Problem 3 – Exercise 10.2 in Mitchell

```
enum shape_tag {s_point, s_circle, s_rectangle };
class point {
  shape_tag tag;
  int x;
  int y;
  point (int xval, int yval)
    { x = xval; y = yval; tag = s_point; }
  int x_coord () { return x; }
  int y_coord () { return y; }
  void move (int dx, int dy) { x += dy; y += dy; }
};
class circle {
  shape_tag tag;
  point c;
  int r;
  circle (point center, int radius)
    { c = center; r = radius; tag = s_circle }
  point center () { return c; }
  int radius () { return radius; }
  void move (int dx, int dy) { c.move (dx, dy); }
  void stretch (int dr) { r += dr; }
};
class rectangle {
  shape_tag tag;
  point tl;
  point br;
  rectangle (point topleft, point botright)
    { tl = topleft; br = botright; tag = s_rectangle; }
  point top_left () { return tl; }
  point bot_right () { return br; }
  void move (int dx, int dy) { tl.move (dx, dy); br.move (dx, dy); }
  void stretch (int dx, int dy) { br.move (dx, dy); }
};
```

```
/* Rotate shape 90 degrees. */
void rotate (void *shape) {
  switch ((shape_tag *) shape) {
    case s_point:
    case s_circle:
      break;
    case s_rectangle:
      {
        rectangle *rect = (rectangle *) shape;
        int d = ((rect->bot_right ().x_coord ()
                  - rect->top_left ().x_coord ()) -
                 (rect->top_left ().y_coord ()
                  - rect->bot_right ().y_coord ()));
        rect->move (d, d);
        rect->stretch (-2.0 * d, -2.0 * d);
      }
    }
}
```

Switch on type tag

Type cast

a) Rewrite this so that each class has a Rotate *method,* and no *tag* field

(in other words, make an OO solution)

Problem 3 - 10.2 a)

```
class Point {
        int x, y;
        void move(int dx, int dy){
                x +=dx;
                y +=dy
        };
        void rotate(){};
};

class Circle extends Point {
        // the inherited point can be the center
        int radius;
        // no new move() or rotate() methods are needed
};

class Rectangle extends Point {
        // the inherited point can be the center, although the original
        // did not have that
        Point topLeft, bottomRight;

        void rotate(){ /* implement rotate here */ };
};
```

Problem 3 - 10.2 a)

```
class Point {
        int x, y;
        void move(int dx, int dy){
                x +=dx;
                y +=dy
        };
        void rotate(){};
};

class Circle extends Point {
        // the inherited point can be the center
        int radius;
        // no new move() or rotate() methods are needed
};

class Rectangle extends Point {
        // the inherited point can be the center, although the original
        // did not have that
        Point topLeft, bottomRight;

        void rotate(){ /* implement rotate here */ };
};
```

Another option would be to add a Shape class at the top of the hierarchy, and letting Point, Circle and Rectangle inherit from this class.

Problem 3 - 10.2 b)

- What if we add a Triangle class? What modifications would be necessary with the original version, and our new version?
  - Original:
    - modify the shape tag enum to include a triangle tag
    - add a new triangle class
    - change the rotate procedure

Problem 3 - 10.2 b)

- What if we add a Triangle class? What modifications would be necessary with the original version, and our new version?
  - New, OO, version:
    - add a new triangle class (with required methods)

```
class Rectangle extends Point {

        Point p1, p2, p3  // the three points defining the triangle

        void rotate(){ /* implement rotate here */ };
        void move(){ /* implement move here */ };
};
```

Problem 3 – 10.2 c)

- Discuss the differences between changing the definition of the rotate method in the original and new (OO) version. (Remember that we have added the Triangle.)
  - Both versions would require invasive changes
    - The Mitchell only to one procedure (the common rotate), while our new solution would require changes to all non-trivial rotate methods.

Problem 4

a) Which of the methods C_equals 1 or SC_equals 1 will be called by the statements below?

- Remember that which overload to call is determined at compile-time, while which override to call is determined at runtime.

```
C  c  = new C();
SC sc = new SC();
C  c' = new SC();
```

c.equals(c)	C_equals 1
c.equals(c')	C_equals 1
c.equals(sc)	C_equals 1
c'.equals(c)	SC_equals 1
c'.equals(c')	SC_equals 1
c'.equals(sc)	SC_equals 1
sc.equals(c)	SC_equals 1
sc.equals(c')	SC_equals1
sc.equals(sc)	equals 2

```
class C {
  ...
  bool equals(C pC) {
    ...                          // C_equals 1
  }
}

class SC extends C {
  ...
  bool equals(C pC) {
    ...                          // SC_equals 1
  }

  bool equals(SC pSC) {
    ...                          // equals 2
  }
}
```

Problem 4

b) Suppose that SC_equals 1 is no longer there.

- Remember that which overload to call is determined at compile-time, while which override to call is determined at runtime.

```
C c      = new C();
SC sc    = new SC();
C c'     = new SC();
```

c.equals(c)	C_equals 1
c.equals(c')	C_equals 1
c.equals(sc)	C_equals 1
c'.equals(c)	C_equals 1
c'.equals(c')	C_equals 1
c'.equals(sc)	C_equals 1
sc.equals(c)	C_equals 1
sc.equals(c')	C_equals1
sc.equals(sc)	equals 2

```
class C {
  ...
  bool equals(C pC) {
    ...                          // C_equals 1
  }
}

class SC extends C {
  ...
  bool equals(SC pSC) {
    ...                          // equals 2
  }
}
```

Problem 5 a)

Write in Java an abstract data type and a class for a data type Date, with year, month and day, and operations *before*, *after* and *daysBetween*.

Abstract data type:

```
class Date {
  int year, month, day;
  Date date(int y, m, d) {.. ; return new Date(...) ; ..}

  boolean static before(Date d1,Date d2) {
    if (d1.year < d2.year) {return true} else
    if (d1.year > d2.year) {return false} else
      if (d1.month < d2.month) {return true} else
      if (d1.month > d2.month) {return false} else
        return d1.day < d2.day;
  };
  ..
}
```

Abstract data type:
meaning of an operation is always the same
operation (operands)

Class:
Meaning of operation might depend on runtime type of object (polymorphism/dynamic dispatch)
object.operation(arguments)

Problem 5 a)

Write in Java an abstract data type and a class for a data type Date, with year, month and day, and operations *before*, *after* and *daysBetween*.

Class

```
class Date {
  int year, month, day;
  Date date(int y, m, d) {.. ; return new Date(...) ; ..}

  boolean before(Date d) {
    if (year < d.year) {return true} else
    if (year > d.year) {return false} else
      if (month < d.month) {return true} else
      if (month > d.month) {return false} else
        return day < d.day;
  };
  ..
}
```

Abstract data type:
meaning of an operation is always the same
operation (operands)

Class:
Meaning of operation might depend on runtime type of object (polymorphism/dynamic dispatch)
object.operation(arguments)

Problem 5 b)

- How would you make the Date class independent of the representation of years, months and days?
  - Abstraction is key!

```
class Month {
  int mAsInt;
  Boolean before(Month m) {
    return mAsInt < m.mAsInt;
  }
};
class Year {/* correspondingly */};
class Day {/* correspondingly */};

class Date {
  Year year; Month month, Day day;
  Date date(Year y, Month m, Day d) {..}

  boolean before(Date d) {
    if year.before(d.year)
      {return true} else
      if month.before(d.month)
       {return true} else
        return day.before(d.day); }
}
```