a) Name 4 kinds of primitive types and 4 composite ones.

Generally, we distinguish between basic and composite types. The former are atomic and built into the language, while that latter are composed out of one or several simpler types. Hence, basic types are types such as • integers, signed and unsigned, of various precisions, including arbitrary precision integers, • floating point numbers of various precisions, decimal numbers (0000.00), and arbitrary precision rational numbers, • integer ranges (1..100), • enumerations (enum colours { Red, Green, Blue, Yellow }), • booleans, • characters, • strings, • the empty type and the unit type, while the composite types include 27 3 Types • arrays, • pointers and references, • functions and procedures, • records and tuples, • unions and variants, • lists and maps or dictionaries.

b) What is a coroutine?

On the language level concurrency manifests in having several independent paths of execution', that is, instead of having a single code location identifying the expression that is to be executed next, there can be several such locations. Each of the expressions pointed to will be executed eventually, but the ordering in which this happens is left unspecified. Such a path of execution is called a fibre of the program. If fibres are executed in parallel, they are also called threads or processes. Fibres that are not executed in parallel are called coroutines. Thus, a fibre is a part of the program with its own control flow. Within each fibre execution is linear, but the program execution can jump between fibres at certain points. Even without parallelism, fibres have the advantage that the program is no longer restricted to a single syntactic nesting and stack discipline. It can use several of them in parallel.

c) What is a monitor?

Monitors A monitor is an abstract data type that is protected by a lock. Each operation on the type first acquires the lock and releases it again upon return.

d) Name two types of problems commonly arising in concurrent systems.

Deadlocks - A deadlock is a situation where there are several fibres, each waiting on an action that can only be performed by one of the other waiting fibres.
Race conditions - A race condition is a bug that is caused by the particular choices and timing of the scheduler.
Starvation - If a fibre is ready but it is never executed because there is always another fibre that goes first, we say the fibre is starving.
Fairness - When several fibres compete for a certain resource, we ideally want them to get access to the resource in equal amounts. This is called fairness.

e) What are the advantages and disadvantages of using multiple inheritance.

- complexity

+ using code of more objects at once
+ adding more power to the language

6. Name three forms of polymorphism and explain what they are.

One can broadly distinguish three different forms of polymorphism.
(i) ad-hoc polymorphism, also called overloading,
(ii) parametric polymorphism as can be found in ML-like languages, and
(iii) subtyping polymorphism as is present in object-oriented languages.

In ad-hoc polymorphism the programmer can define several versions of a function. Which of them will be selected when the function is called will depend on the types of the supplied arguments. A typical example are arithmetic operations which in many languages are defined both for integers and floating point numbers.
1 + : int -> int -> int
2 + : float -> float -> float

In parametric polymorphism we allow type expressions to contain type variables. For instance, we can specify the type of the map function as map : (a -> b) -> list(a) -> list(b) with two variables a and b. This is a simple and quite clean extension of the type system with few drawbacks. But is is less flexible than ad-hoc polymorphism. Most of the functional programming languages have adopted this version of polymorphism.

This kind of polymorphism is based on the subtyping relation. We say that a type a is a subtype of b if every value of type a can be used where a value of type b was expected.

7. Explain how exceptions and exception handling work.

This exception mechanism works similarly to the break-statement of imperative languages. But instead of jumping out of loops, i.e., out of a nested static scope, it allows the program to jump out of nested function calls.

Exceptions can be implemented using continuations. Every function gets as an additional argument a continuation to call when raising an exception. Acatch statement uses letcc to create such a continuation.

Although they can be considered as a generalised break statement, the destination of an exception is determined dynamically by the sequence of function calls and not statically by the syntactic structure of the program.

<u>What is binding?</u>

Apart from making code more readable and easier to write, local definitions can also be used to improve performance. If a complicated expression is used in several places, we can use a let-binding to evaluate the expression only once and then refer to its value via the corresponding identifier. For instance, if we want to rotate a vector, we only need to compute the sine and cosine once.

In general the association of names in a program with the entities they refer to is called binding

Different versions of parameter passing

Call-by-value is the standard mechanism for in-mode parameters. When calling a function, the argument values are passed as copies to the function body. Modifications of the copies do not affect the originals. This is a very safe method that avoids any confusion cause by unexpected modifications. The disadvantage is that it can be inefficient if large objects are passed in this way.

Call-by-result is the analog of call-by-value for out-mode parameters. No value is passed during the function call. Instead, when the function returns, the current contents of the variable corresponding to the parameter is copied back to the argument, which must be an l-value. Callby-result has the same advantages and disadvantage as call-by-value.

Call-by-value-result/call-by-copy/call-by-copy-result combines call-by-value and call-byresult for in/out-mode parameters. The argument value is copied to the parameter when the function is called and copied back, when it returns.

Call-by-reference is a more efficient version of call-by-value-result. Instead of copying the value back-and-forth,its address is passed to the function. Every Modification inside the function directly affects on the original l-value. This is very efficient, but can create aliasing problems.

In call by name, you substitues in the body of the function any references to arguments by their code used during the call. Then, evaluating the body, you will evaluate the arguments. foo(bar()) with foo(arg) { return arg; } will be evaluated as foo(arg) { return bar(); }

Call-by-need is an optimised version of call-by-name useful for in-mode parameters. It is the standard calling convention used in lazy functional languages like Haskell. Here, after the first evaluation of a passed argument expression, the result is stored, so subsequent uses of the parameter do not need to evaluate the expression again. Of course, this only works if the argument expression has no side effects.

Static vs dynamic scoping

When invoking a function,static scoping evaluates the function body in the scope of the function’s definition, while dynamic scoping uses the scope of the function’s caller.

Examples of languages using dynamic scope are: the original Lisp, Emacs Lisp, TeX, Perl

Example of type inference

Example
let twice(x) { 2 * x };
We start by associating a type variable with every subexpression.
2 : α
x : β
* : γ
2*x : δ
fun (x) { 2*x } : ε
We already know that α = int and γ = int -> int -> int .
By looking at each subexpression in turn, we obtain the following additional equations.
2*x : α = int , β = int , δ = int fun (x) { 2*x } : ε = β -> δ Solving them we obtain. ε = β -> δ = int -> int .

Example
let compose(f,g) { fun (x) { f(g(x)) } }
Again we start by associating type variables with subexpressions.
$x : \alpha$
$f : \beta$
$g : \gamma$
$g(x) : \delta$
$f(g(x)) : \varepsilon$
fun (x) { f(g(x)) } : ξ c
ompose : η
The equations are
$\gamma = \alpha \to \delta$
$\beta = \delta \to \varepsilon$
$\xi = \alpha \to \varepsilon$
$\eta = \beta \to \gamma \to \xi$
which lead to the solution $\eta = \beta \to \gamma \to \xi = (\alpha \to \delta) \to (\delta \to \varepsilon) \to \alpha \to \varepsilon$.

Name three mechanisms that can be used to implement mutual exclusion and explain how they work
- Lock
- Semaphores – generalization of lock -> counter with increase/decrease
- Futures - Futures are a simple synchronisation mechanism where we can evaluate a given expression in parallel and wait for the result. They work like a channel that can only be used a single time.

What is subtyping

A type s is a subtype of a type t if values of type s can be used everywhere a value of type t is expected. As with type equivalence there are two different approaches to implement subtyping: structural and by name. In languages like Java where subtyping is defined by name, the programmer has to explicitly declare if one object type is to be a subtype of another. In languages with structural subtyping on the other hand, a type s is automatically a subtype of all types that are more general than s. This means that, if s and t both are object types, then s will be a subtype of t if s supports all the methods of t. For instance, if we have defined a class of shapes with methods draw, move, and box and a subclass of rectangles with an additional method area, then the rectangle class is a subtype of the shape class.

Variant type

Can contain any type. In dynamic languages this is basically everything

Continuation

We need (i) a data structure storing at what place in the program we are and (ii) a way to use this data to resume the program at that point. To resume the computation of the program from an arbitrary point we need to know
• where in the program we are, i.e., what the last evaluated expression was,
• what the result of this expression was, and
• what the values of the local variables were. We can store this information as a function that, given

the result of the last expression, continues the program from this point. Such a function is called a continuation.

```
 let f () {
      let u = input("first: ");
      let v = input("second: ");
      process(u,v);
 };
```

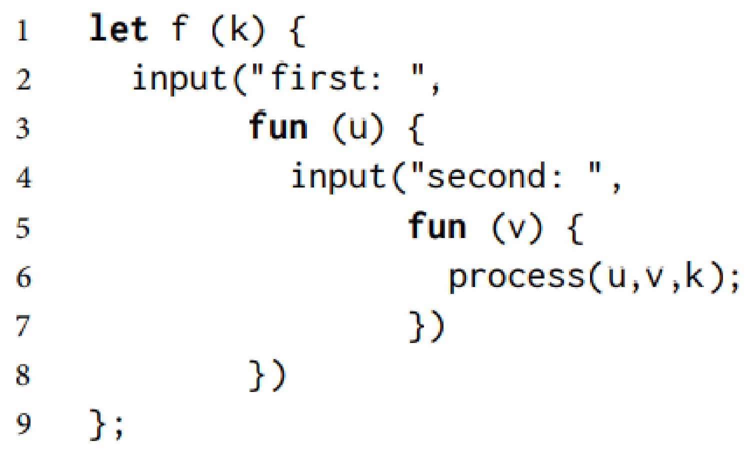
For instance, in the above example the continuation after having read the first input is

```
fun (u) {
  let v = input("second: ");
  process(u,v);
};
```

The continuation after the second input is

```
fun (v) {
  process(u,v)
};
```

In order to prepare our program for usage with a web-server, it is useful to translate it into a form that makes these continuations explicit. This form is call continuation passing style, CPS for short. In this form, every function takes an additional argument k that takes the continuation to be called when the function wants to return. Our example now looks as follows.



```
let f (k) {
  input("first: ",
        fun (u) {
          input("second: ",
                fun (v) {
                  process(u,v,k);
                })
        })
};
```

<u>1. Which programming languages is a common ancestor of Pascal and C?</u>
- Algol 60

<u>2. Name the language, which was intended to replace, at the same time, Fortran, COBOL and Lisp.</u>
- PL/I

<u>3. Which language (created 1980) developed the concepts of classes and OOP, which First appeared in SIMULA 67?</u>
- SmallTalk

<u>4. Which two control structures are su-cient to express any algorithm described by a Flow chart?</u>
- if and loop

<u>5. State the simple rule for combining positional and keyword parameters in a single function.</u>
- once a keyword parameter appears, all following ones must be keyworded

<u>6. What is the main disadvantage of using interfaces in Java (compared to using class inheritance)?</u>
- no code reuse

<u>Explain the concept of monitors   what are they used for, how they work, what are their advantages compared to semaphores?</u>

A Monitor is an object designed to be accessed from multiple threads. The member functions or methods of a monitor object will enforce mutual exclusion, so only one thread may be performing any action on the object at a given time. If one thread is currently executing a member function of the object then any other thread that tries to call a member function of that object will have to wait until the first has finished.

A Semaphore is a lower-level object. You might well use a semaphore to implement a monitor. A semaphore essentially is just a counter. When the counter is positive, if a thread tries to acquire the semaphore then it is allowed, and the counter is decremented. When a thread is done then it releases the semaphore, and increments the counter.

<u>Give an example of a: • unary assignment statement: • compound assignment statement:</u>
– Unary: count++; Compound: a += b

<u>Explain the difference between coercion and (explicit) type cast</u>
Coercion: implicit type conversion, for compatible types

<u>What are the differences between tasks and subroutines?</u>

• a task can be executed implicitly
• the unit executing a task may not necessarily wait for it to finish
• after a task is finished, the execution can continue from a different point than the point of task execution