In questo documento verranno inserite tutte le domande che il prof fa al inizio di ogni lezione, prese della registrazioni. Le risposte, dove presenti, prendetele con le pinze perchè provengono dai colleghi e non da Coen, secondo me a volte CSC si accontenta. Io comunque per comodità le ho trascritte come pronunciate.

**26/02**

**Cos'è un paradigma?** Un paradigma è una risposta ad una domanda che ci da una visione totalmente diﬀerente da altre visioni da generare prospettive tecniche, idee e metodologie diﬀerenti.

**Abbiamo fatto esempi di paradigmi in informatica, i primi di cui abbiamo parlato sono i paradigmi in calcolabilità. Quali sono?** Funzionale (si rifà al lambda-calcolo), paradigma imperativo e logico.

NOTA CSC: Erlang fa parte del paradigma ad attori, che è un paradigma composito in cui, almeno nel caso di Erlang, una delle componenti è il paradigma funzionale, anche se ci sono altri linguaggi ad attori che adottano il paradigma ad attori su linguaggi imperativi.

Abbiamo citato 3 paradigmi. Quello imperativo da cosa è caratterizzato? Da uno stato residente in memoria e dalle operazioni che modiﬁcano questo stato per compiere una computazione

**Il paradigma funzionale invece?** Il paradigma funzionale si basa principalmente sul lambda-calcolo e sull'immutabilità delle strutture dati. E' un paradigma che basandosi sul lambda-calcolo sfrutta la riscrittura dei termini.

**NOTA CSC:** nel paradigma funzionale l'idea è che non si modiﬁca niente, bensì si riscrivono le espressioni in altre espressioni, ﬁn quando l'espressione iniziale, che sono i dati in input ed il programma in input, vengono trasformati nell'espressione ﬁnale, che contiene l'output.

**Il paradigma logico invece?** Si occupa di trovare una relazione tra l'output e l'input. E' la ricerca di una prova in cui sono forniti assiomi logici.

**L'altra parola nel titolo del corso è "Emergenti". Perché parla di emersione?** Emergenti non indica linguaggi degli ultimi anni, perché vediamo anche linguaggi del secolo scorso, ma indica come paradigmi non usati nel tempo tornino ad essere utili per aﬀrontare scenari moderni, quindi li studiamo perché potrebbero essere utili al giorno d'oggi.

L'evoluzione dei linguaggi di programmazione è caratterizzata da periodi abbastanza di stasi, in cui ci sono linguaggi consolidati, che hanno più o meno le stesse feature, e che più o meno risolvono i problemi e poi ad un certo punto ci sono dei periodi in cui c'è un fattore scatenante, per cui serve integrare nei LdP qualche feature in più, c'è un certo fermento in cui nascono tanti linguaggi, di qui qualcuno diventa mainstream e gli altri mainstream tendono ad adottare i costrutti di questi nuovi linguaggi per non essere scartati. Abbiamo citato quali sono i due motivi scatenanti, quello precedente e quello ancora in corso. Quali sono? Uno era l'invasione dei core, il loro aumento esponenziale, che richiede forzatamente programmazione concorrente, e l'altro è l'ondata di Java, C#, l'introduzione di programmi che dovessero gestire server di rete, prima programmi che interagissero con server erano in quantità minima. Scala ha usufruito della tendenza alla cloudiﬁcazione.

**L'evento scatenante più recente è la nascita dei core e quindi l'esigenza di fare programmazione concorrente. Perché abbiamo computer con un numero sempre maggiore di core?** Perché abbiamo raggiunto il limite in cui non riusciamo ad aumentare la potenza di un singolo core, quindi per avere più potenza di calcolo aumentiamo il numero di core.

**Finita l'intro. Abbiamo iniziato a parlare di programmazione ad attori? Un attore è fatto di tre componenti. Quali sono?** PID, behavior e mailbox

**Uno degli ingredienti è il PID. Cos'è e cosa posso fare con un PID?** Il PID è un identiﬁcatore univoco dell'attore, è un tipo di dato astratto che non identiﬁca un indirizzo in particolare. Possiamo solo confrontare i PID tra loro ma non molto altro.

**CSC: Non è in nessun modo un indirizzo, questo fa sì che i linguaggi di programmazione hanno iniziato tutti a diventare concorrenti per supportare i core ed Erlang è stato l'unico linguaggio a cui non è stata apportata nessuna modiﬁca perché, visto che il PID non identiﬁca un indirizzo, questo fa sì che il fatto che l'attore sia in esecuzione su un altro core o su una macchina remota è del tutto invisibile e trasparente al programmatore.**

**I PID possono essere solo confrontati e poi abbiamo visto operazioni speciﬁche, ad esempio dato un PID posso inviare un messaggio**

**Cosa si può dire sulla mailbox invece?** La mailbox è una coda di messaggi non acceduta con la politica FIFO, ma con un accesso OutOfOrder, perché si adatta a scenari distribuiti e l'ordine di ricezione dei messaggi non è detto che sia quello giusto.

**Abbiamo detto un'altra caratteristica di questa mailbox, che è legata anche al tipo di message passing nella programmazione di Erlang. Qual è la dimensione di questa coda? Inﬁnita Perché?** L'operazione di send che caratteristiche ha? La coda è inﬁnita e i messaggi sono asincroni. L'invio di in messaggio è sempre asincrono, il mittente non attende nessun feedback da parte del ricevente, pertanto il ricevente può continuare a ricevere messaggi a cui non sta rispondendo, perché non li sta processando e la coda inﬁnita permette di gestire la programmazione asincrona. Se invece avessimo una coda bounded (es 100), i primi X messaggi possono essere gestiti sia in maniera sincrona che asincrona, ma arrivati al 100esimo il 101 si deve fermare perché non c'è spazio nella coda.

**Questo inﬂuisce su un altro paradigma, il paradigma di trasmissione, che può essere sincrono o asincrono.**

**Gli attori hanno memoria condivisa?** No, perché altrimenti avremmo problemi di gestione, i.e. dovremmo implementare lock, mutex e aﬃni, che è complesso da gestire in un sistema distribuito. (i lock si portano dietro starvation, deadlock, ... Se un processo fa lock e poi fallisce non si sa come comportarsi, stessa cosa sarebbe la comunicazione sincrona, perché il mittente deve aspettare il ricevente e si potrebbero creare deadlock, starvation,..) 20:52

**Terzo ingrediente citato è il behavior. Cos'è il behavior?** E' una mappa da messaggi in azioni, che possono essere comptazione interna, invio di messaggi in maniera sincrona, creazione di altri attori. Un behavior termina in un altro behavior e l'attore si sospende in attesa di un altro messaggio.

**La programmazione ad attori si basa sul concetto di behavior. E' una classe più ampia, di un paradigma più ampio. Qual è?** Il paradigma event-driven, paradigma in cui nella maggior parte del tempo i processi sono sospesi in attesa di messaggi e reagiscono a messaggi o ad eventi

23:50

**Erlang è considerato il linguaggio di riferimento per la programmazione ad attori. Nasce come linguaggio logico distribuito per implementare switch e hw telefonici e diventa un linguaggio funzionale distribuito e riemerge la programmazione ad attori, deﬁnita in precedenza. Erlang introduce altre caratteristiche oggi considerate parte della programmazione ad attori, che hanno a che fare con la gestione dei fallimenti. Qual è la politica di gestione dei fallimenti di Erlang?** Quando un attore fallisce è distrutto, ma gli attori che stanno collaborando a portare a termine un compito sono legati tra loro e se uno è distrutto tutti quelli che collaboravano con lui sono distrutti, perché magari, se collaboravano con lui, aspettavano messaggi da lui, quindi tantovale uccidere gli attori che stavano collaborando. Un gruppo di attori ha un supervisore, che li ha creati e li supervisiona e se muoiono li fa ripartire. Questo è semplice in Erlang, perché non c'è uno stato che viene modiﬁcato, quindi il padre ha a disposizione tutti i dati (non modiﬁcati) per ricreare i ﬁgli allo stesso modo.

**Un ingrediente necessario dev'essere l'eﬃcienza in creazione e distruzione di attori. Erlang è eﬃciente nella gestione degli attori?** 27:00 Dal punto di vista della performance in erlang bisogna distinguere i thread kernel ed i thread del linguaggio. Ogni attore ha un thread del linguaggio, che poi questo non è detto che sia eseguito in maniera concorrente, perché la BEAM, la VM di Erlang, crea un kernel thread per ogni core, ed ogni attore, quando lo deve eseguire, viene eseguito su questi kernel thread.

Erlang è uno dei primi linguaggi ad essere estremamente eﬃciente nella gestione dei thread al punto da poter gestire milioni di attori contemporaneamente. Questo è possibile perché un attore ha una rappresentazioneestremamente piccola (~300 word per deﬁnire un attore) ed il context switch è quindi estremamente veloce nel salvataggio ed il ripristino dello stato

**02/03**

**Abbiamo iniziato a parlare di Erlang come linguaggio funzionale, prima abbiamo introdotto la programmazione logica con elpi ed abbiamo introdotto le caratteristiche dei LdP funzionali. Abbiamo parlato di programmazione funzionale, non c'è una deﬁnizione unica di cosa sia un programma funzionale. Ci sono delle caratteristiche che in genere i linguaggi funzionali hanno e gli altri hanno solo in parte, anche se negli ultimi anni queste caratteristiche stanno entrando un po' in tutti i linguaggi. Quali sono?** Devono avere due caratteristiche in particolare: funzioni higher order e purezza, ossia assenza di side eﬀects

**Le funzioni higher order cosa sono?** Per essere higher order le funzioni devono poter essere prese in input, date in output e deﬁnite localmente

**Le funzioni possono essere un oggetto di prima classe. Quand'è che una funzione è un oggetto di prima classe?** Quando sono usate in maniera libera come qualsiasi altro valore presente all'interno del linguaggio. Per esempio un LdP con funzioni di ordine superiore in cui non posso creare un array di funzioni, sarebbe un linguaggio di programmazione di ordine superiore, ma le funzioni non sarebbero oggetti di prima classe, perché gli array sarebbero speciali e non ammetterebbero al loro interno funzioni.

**Per quale motivo è interessante ed utile avere funzioni di ordine superiore?** Perché ci permette di avere una genericità all'interno delle funzioni, è possibile creare delle funzioni che siano degli operatori di controllo di livello più alto. In un frammento di C devo leggere la totalità del while per capire cosa stia facendo, in un linguaggio di ordine superiore implemento delle funzioni di ordine superiore, che mi permettono di capire meglio cosa sto facendo. Ad esempio, se sto invocando una map su una lista, so già che quella map itera su una lista per creare un'altra lista. Se sto invocando una fold sto computando una funzione di tutti i valori in una lista. Grazie alle funzioni di ordine superiore, scrivo del codice, che è molto più vicino al dominio del problema rispetto al dominio della soluzione.

**Quali sono i linguaggi di programmazione che ho citato, che lei conosce, che non hanno funzioni di ordine superiore?** C non è un linguaggio di ordine superiore e Pascal permette di deﬁnire funzioni in funzioni ma non è un linguaggio di ordine superiore**. Più precisamente cosa non permette di fare C e cosa non permette di fare Pascal?** In C posso solo prendere in input puntatori a funzione, che devono già esistere al top level, e posso dare in output dei puntatori a funzione, ma non posso deﬁnire una funzione dentro un'altra, posso solo passare in giro funzioni pre-esistenti. Abbiamo visto invece l'altro giorno una funzione, un adder, a cui se passavo 3, mi dava una funzione che aggiunge 3, se passavo 5, mi dava una funzione che aggiunge 5, ... In C questo non riesco a farlo. In Pascal potevo creare funzioni una dentro l'altra, ma non potevo passarle in giro, anche quello non è ordine superiore.

**Cosa si intende per purezza?** Intendiamo che non abbiamo side eﬀect, non possiamo modiﬁcare il valore di una variabile in memoria

**Ci sono altri side eﬀect che nei linguaggi completamente puri sono inibiti? O più in general cos'è un side eﬀect?** Un side eﬀect è tutto ciò che non ha a che fare con il behavior matematico di una funzione, il rapporto tra input ed output. Una funzione su uno stesso input mi restituisce sempre lo stesso output e non altera nient'altro. **I LdP oltre a mutare le variabili possono fare qualcos'altro. In genere cosa possono fare?** I side eﬀect sono una qualunque modiﬁca allo stato globale. Un esempio sono tipo le stampe, tutto ciò che è l'I/O. Se invoco due volte una funzione questa può darmi due volte un output. Sono pochi i LdP completamente puri rispetto ai side eﬀect, Haskel è un tipico esempio, molti LdP della famiglia funzionale possono essere più o meno puri, tutti questi ammettono l'I/O ma restringono l'accesso ai dati e la mutabilità. Quasi tutto è immutabile, ma ciò nonostante possiamo fare I/O

**Abbiamo iniziato a vedere Erlang e abbiamo parlato di tipi di dato atomici e composti. Ti ricordi quali sono i tipi di dato atomici di Erlang?** Boolean, int, ﬂoat, string, atomi (possono solo essere confrontati tra loro), PID degli attori (possono solo essere confrontati), i cookies (sono dei nounces, legati ad un messaggio per evitare attacchi), le porte

Abbiamo parlato anche di tipi di dato non atomici. Erlang ne ha un numero limitato, quali sono? Ci sono le tuple, usate anche per inviare i messaggi agli attori se sono etichettate, le liste e le mappe, ed i record, che sono in realtà zucchero sintattico, che in realtà sono implementate tramite tuple.

**Abbiamo iniziato a parlare dei costrutti del LdP Erlang, ed abbiamo introdotto la nozione di pattern e pattern matching. Cos'è il pattern e cos'è il pattern matching? In quali costrutti possiamo usare i pattern ed i mattern matching in Erlang?** Ne abbiamo visti 3 per ora La deﬁnizione di funzione, quando deﬁnisco una funzione, specie al top level, un ramo della funzione è eseguito solo se l'input fa match con quel pattern, ed abbiamo visto che è possibile deﬁnire una funzione per casi sulle forme dell'input, scrivendo tante volte f(patter)->qualcosa, separati da ;

Gli altri due costrutti uno era l'uguale di erlang, che non è l'assegnamento ma il pattern matching. x=qualcosa lega ad x il valore ma se ho una tupla assegna ad x il primo valore e ad y il secondo

Il terzo costrutto è le receive dei messaggi, quando scrivo receive scrivo receive (pattern, behavior)

TODO continuare dal minuto 21

**04/03**

**Diﬀerrenza programmazione agenti ed attori.** Ci sono varie forme di diﬀerenze sulla base della letteratura. La programmazione ad agenti è un caso particolare di quella ad attori dove si vincola la forma dello stato ed anche i messaggi.

Quest'anno non ho ancora parlato di questa diﬀerenza, c'è già il post su IOL dell'anno scorso (C&P). Mentre la programmazone ad attori è molto precisa come deﬁnizione (un po' storicamente perché fu precisa a livello accademico, un po' perché Erlang l'ha ﬁssata essere quella cosa lì) per la programmazione ad agenti ci sono n-mila deﬁnizioni di cosa sia un programma ad agenti già in letteratura e non concordano. Quello a cui fa riferimento il post è che in generale molte delle proposte di programmazione ad agenti cercano di ricalcare l'idea di andare a modellare un agente ﬁsico in un mondo ﬁsico (es robot)

che deve computare in base ad un suo obiettivo per raggiungerlo e questo obiettivo ha a che fare con la modiﬁca dello stato del mondo ﬁsico. L'agente in questione agisce nel mondo ﬁsico assieme ad altri agenti. I linguaggi di programmazione ad agenti sono pensai per catturare queste situazioni. Come il modello ad attori non hanno memoria condivisa (es. auto a guida autonoma) e comunicano attraverso message passing (questa però non è l'unica forma di comunicazione, in altri paradigmi ad agenti non vi è nemmeno message passing diretto, bensì c'è uno spazio delle tuple, i.e. l'idea è che se posso modiﬁcare il mondo ﬁsico e tu puoi leggere il mondo ﬁsico io posso lasciarti dei messaggi nel mondo ﬁsico. L'idea è avere uno spazio in cui posso lasciarti dei messaggi e tu puoi prelevarli. Un messaggio può essere qualunque cosa, un record contenente degli atomi, io posso depositare questo messaggio e chiunque può prelevarlo. A diﬀerenza di Erlang dove devo conoscere il PID del ricevente in questo tipo di modello inece nessuno conosce gli altri a priori.) Il secondo punto è molti di questi linguaggi basati sul paradigma ad attore vincolano i messaggi perché vincolano lo stato dell'agente. Lo stato ha un obiettivo che può essere codiﬁcato in un dominio predeﬁnito, c'è una descrizione molto più limitata di quello che può essere un stato di un agente e i messaggi che si possono scambiare gli agenti non sono messaggi generici ma sono preﬁssati. L'idea è che tu comunichi con agenti che non condividono il tuo codice, quindi occorre ﬁssare a priori messagig che si possono scambiare e la loro semantica in modo che agenti che implementano codici complretamente diﬀerenti possono comunque interagire. Idea che cercava di rendere: il linguaggio vincola al protocollo di comunicazione

**Ma quindi negli attori non si vincola la semantica dei messaggi?**

Un attore può mandare ad un altro PID qualunque messaggio e l'altro attore può interpretare il messaggio come gli pare.

Quando si parla della deﬁnizione di linguaggio puro ci riferivamo solo ai funzonali? Perché negli imperativi il side eﬀect è quasi necessario.

La purezza fa riferimento a qualunque side eﬀect, in generale c'è la mutabilità della memoria e poi ci sono tutti gli altri side eﬀect. Se uno chiede la purezza come totale assenza di side eﬀect essenzialmente rimane Haskell.

Se togliamo i side eﬀect per esempio di I/O e teniamo la mutabilità abbiamo tanti linguaggi funzionali in più, che sono puri in quel senso ed abbiamo anche librerie per linguaggi che puri non nascono per fare programmazione pura in questo senso, es. Java e Scala, che hanno la libreria Cats per chi vuole programmare alla Haskell all'interno di Java (con strutture dati immutabili).

Alla domanda se ci sono linguaggi puri (almeno dal punto di vista della mutabilità) e che non sono funzionali, la risposta è no. Nel momento in cui un programma diventa puro devo abbandonare il paradigma imperativo (i cicli non servono più a niente) ed a quel punto diventiamo funzionali. Un po' di immutabilità sta entrando in tutti i LdP, un po' perché in un ambito concorrente l'immutabilità ci salva, perché evita le race condition per deﬁnizione, quinid i linguagi imperativi permettono di avere le keyword per dire che certi dati sono immutabili e quindi si introduce un po' di immutabilità nei linguaggi imperativi. Spesso questa immutabilità è superﬁciale, per esempio potrei dire questo record è immutabile ma se quei campi sono dei dati composti, es. degli oggetti, loro possono essere mutabili, quindi non ho la garanzia dell'immutabilità completa del dato.

**Ho risposto?** Io direi decisamente sì

**11/03**

1. Come si fa in un libreria C una funzione che lavora su tipi di dimensione arbitraria?
   1. quando si prende in input un dato che può avere un tipo arbitrario, si prende in input un puntatore al inizio del area di memoria del dato e una dimensione del dato
2. Come faccio a tempo di compilazione a sapere quanto è grande un dato in C?
   1. sizeof
3. La sizeof è una funzione o un operatore del linguaggio?
   1. un operatore del linguaggio, perchè sarebbe impossibile definirla in liberia.
4. Quali sono possibili svantaggi di questo approccio di C?
5. In cosa in consiste la monomorfizzazione?
6. Quali sono i vantaggi della monomorfizzazione?
7. Quali sono invece gli svantaggi?
   1. più lento, avere tanto codice causa uno swap continuo tra ram, cache e registri.
8. In cosa consiste la rappresentazione uniforme dei tipi di dato?
   1. Rappresentiamo tutti i dati con una parola, che può essere il valore in sè o un puntatore nel caso il dato sia più grande di una parola.
9. Perché risolve i problemi dei metodi descritti prima?
10. Differenze boxed e unboxed.
11. Quello che abbiamo detto ora sarebbe sufficiente se il linguaggio di programmazione non avesse ulteriori esigenza, abbiamo visto 2 esigenze, la prima per implementare la garbage collection in un certo modo, il runtime che implementa la garbage collection ha bisogno di un'informazione, qual’è?
12. L’altra esigenza linguistica qual’è? (erlang ce l’ha, ocaml no)
13. In cosa consiste il tag?
14. Quando è grande il payload?
    1. Dipende dalla dimensione del tag
15. Questo suggerisce che bisogna minimizzare la dimensione del tag, qual’è una strategia per farlo?
    1. Abbiamo tag a dimensione variabile.
16. Quali problemi comporta per le operazioni aritmetiche questo approccio?
    1. Va a cambiare come dovremo fare le operazioni base per avere un risultato corretto
17. C’è un ultimo problema di cui abbiamo parlato quale?
    1. il problema del indirizzamento della memoria, anche togliendo un bit, possiamo indirizzare la metà della memoria. Soluzione: indicizziamo solo le celle pari.

**18/03**

1. Cos’è lo stack delle chiamate e il record di attivazione?
2. Cosa si intende per chiamata di coda?
3. Quando una funzione è tail ricorsiva?
   1. quando tutte le chiamate ricorsive al suo interno sono di coda
4. Perchè se Erlang non avesse l'ottimizzazione di cosa il linguaggio sarebbe inusabile?
5. Non tutte le funzioni nascono come di coda, quali sono i modi per trasformare una funzione non tail ricorsiva, tail ricorsiva?
   1. stack e CPS
6. qual è un vantaggio in certi linguaggi di programmazione di emulare a mano uno stack e perchè in Erlang non è necessario?
   1. Erlang implementa lo stack con un array estensibile salvato nello heap, non più un come nei vecchi linguaggi in cui ad ogni chiamata ricorsiva era assegnata solo una pagina in memoria (che poteva finire). In questo modo Erlang può creare più thread.
7. Qual è l’idea dietro alla CPS translation?
   1. Dare a ogni funzione definita una sua continuazione in modo da ottimizzare le funzioni e renderle tutte di coda perchè a quel punto ognuna una chiamata di funzione alla fine (letterali parole di stac)
8. Cos’è una continuazione?
   1. e’ una funzione che prende il risultato di un altra funzione e poi fa altre cose (letterali parole di stac e il prof ha detto “esatto!”)

**25/03**

1. Il costrutto receive è bloccante, un processo potrebbe rimanere bloccato per sempre se chi gli manda il messaggio non è più disponibile perchè c’è una disconnessione di rete. Come si può ovviare a questo fatto?
   1. Il costrutto receive ha una forma sintattica per prevedere questo timeout.
2. Quali sono i valori ammessi dal costrutto after?
3. Può succedere che la receive abbia pattern multipli, quale semantica si utilizza? come viene determinato quale messaggio per prima viene elaborato?
   1. In base all'ordine del invio dei messaggi. Se non c’è un pattern che fa match si passa al messaggio successivo.
4. A cosa serve register?

**01/04**

Non ha fatto domande

**06/04**

1. Cosa servono le porte?
   1. Ci permettono di fare binding e quindi interfacciarsi con programmi scritti in altri linguaggi.
2. Come uso le porte?
   1. open port
3. Quale è il tipo di dato che viene restituito da open port?
4. Quale sono i 3 possibili scenari di gestione della heap in un sistema concorrente?
5. Qual’è il vantaggio del sistema a heap separato e heap condiviso?
6. Come fa Erlang a gestire il multicore?
7. Quali sono le problematiche che si hanno se si passa a un sistema distribuito?
8. C’è un altro tipo di dato che quando viene inviato/ricevuto cambia, qual’è?

**8/04**

1. Qual'è la differenza tra multitasking preemptive e collaborativo?
2. Qual’è il meccanismo che si utilizza per passare il controllo nel multitasking preemptive nei linguaggi interpretati?
3. E nei linguaggi non interpretati?
4. Quali sono i vantaggi e svantaggi del multitasking preemptive implementato con gli interrupt dal sistema operativo?
5. Quali sono i vantaggi e svantaggi invece quando è implementato in una virtual machine come la BEAM?
6. Quali sono i metodi che posso utilizzare su un ledger?
7. Come si chiamano le celle di un ledger?
8. Quali sono usi possibili delle transazioni?
9. Cos’è uno smart contract?
10. Per cosa si utilizzano gli smart contract?
11. Rispetto a un programma tradizionale, uno smart contract può fare una cosa in più, cosa?
    1. Può costringere gli esseri umani ad avere certi comportamenti
12. Quali sono gli svantaggi nel implementare un ledger in modo centralizzato?
    1. E’ un single point of failure
13. Qui entra in gioco la blockchain che precisamente che tipo di implementazione del ledger è?
    1. *Distribuita, precisamente peer to peer*.
14. Qual’è il problema principale nella gestione di un ledger peer-to-peer?
15. Come si “risolve”?
16. Cosa intendo per consistenza approssimata?
17. Cosa fa un nodo se si accorge della presenza di un fork?
18. Che conseguenza ha sul utilizzatore del ledger il fatto che possa divergere?
19. Per cosa viene utilizzato l’algoritmo di gossiping?
20. E’ un algoritmo push o pull?
21. Quale problematica crea il fatto di scegliere sempre la catena più lunga?
22. Come si risolvere questa problematica?
23. Questo però crea un altro problema, se è costoso mantenere la struttura, come faccio sì che gli utenti onesti partecipino?
24. A monte di tutto questo, rimangono dei problemi. Quali struttura dati si utilizza per far si di accorgersi se qualcuno ha modificato la catena.

**15/04**

1. Cosa intendiamo per future o promise?
2. Quali sono i metodi che definiscono una feature?
3. Quale è la semantica che un feature deve implementare?
4. Cosa si fa in presenza di side effect?
5. La proprietà migliore che hanno i future ha a che fare con il ragionamento della correttezza del codice, perché?
   1. Possiamo leggere il codice in modo sequenziale
6. Quali sono le 2 semantiche dei future?
7. Nel caso l’esecuzione avvenga nel thread corrente ma solo al momento della yield, che anche nome prendono i future?
   1. Chunk, e servono per avere computazioni lazy in linguaggi eager
8. Per quanto riguarda la semantica concorrente, questa a un grande vantaggio ed è anche il motivo per cui i future sono stati riscoperti recentemente, qual è questo vantaggio?
   1. riduce il traffico di rete
   2. più facile parallelizzazione, viene fatta in automatico
9. Qual’è stata l’ultima riscoperta dei future? Da dove viene?
10. Che differenza c’è tra promise e future?

20,22 e 27 non ha fatto domande

**29/04**

1. Che cos’è una riga?
   1. Una riga è una sequenza di coppie (tag,tipo)
2. Quali sono 3 costrutti di programmazione che utilizzano le righe?
   1. le mappe? le mappe è difficile, solitamente i linguaggi di programmazione non hanno mappe generiche in cui in ogni atomo in input associamo un tipo diverso in output spesso le mappe mappano per esempio tutte le stringhe in interi. Quindi non sono le mappe.
   2. Gli ADT. I record. Le classi. Le Union del C.
3. Cosa si intende per polimorfismo di riga?
   1. E’ un tipo particolare di riga in cui quantifico sulle righe e le varianti polimorfe sono un applicazione di questo. Posso dire “per ogni riga succede qualcosa”. I tipi li combino andando a utilizzare il costruttore di tipo, posso fare il prodotto. Le righe le combino facendo l’unione / intersezioni / concatenazioni di righe etc...o ponendo dei vincoli.
4. Cosa dice l’expression problem?
5. Perchè è importante l’expression problem?
6. A riguardo della soluzione all’expression problem che abbiamo dato abbiamo scomposto l’ipotesi del mondo chiuso in due ipotesi differenti, quali erano?