1. El Cálculo λ como fundamento del razonamiento computacional

Pregunta: ¿Cómo puede el Cálculo λ servir no sólo como modelo de funciones, sino también como base formal para razonar sobre la corrección de programas?
Búsqueda: *lambda calculus proof systems, logical foundations of computation, Curry–Howard correspondence, typed lambda calculus*
Programa: Implementar un pequeño intérprete de Cálculo λ con reducción β paso a paso y mostrar cómo una reducción preserva el significado lógico.

2. Expresiones lambda y composición funcional avanzada

Pregunta: ¿Cómo se construyen programas complejos combinando pequeñas expresiones lambda mediante composición y aplicación parcial?
Búsqueda: *function composition, currying, partial application, combinatory logic*
Programa: Implementar un conjunto de transformaciones sobre listas (por ejemplo, normalización de texto) usando únicamente composición ((.)) y lambdas anónimas.

3. Máquinas abstractas y semánticas de evaluación

Pregunta: ¿De qué manera las máquinas abstractas (SECD, CEK, Krivine) permiten modelar distintas estrategias de evaluación y optimización en lenguajes funcionales?
Búsqueda: *CEK machine Haskell, SECD machine, Krivine machine, abstract machine semantics*
Programa: Implementar una versión reducida de la máquina CEK con traza de evaluación para entender cómo maneja los ambientes y continuaciones.

4. Ambientes de evaluación y cerraduras

Pregunta: ¿Cómo influyen las cerraduras (closures) y los ambientes en la implementación de funciones de orden superior?
Búsqueda: *closures environment model, lexical closures, implementation of higher-order functions*
Programa: Simular la creación de un ambiente en el que se definan funciones con variables libres, mostrando cómo el ambiente se conserva dentro de la cerradura.

5. Alcance estático y dinámico en lenguajes mixtos

Pregunta: ¿Por qué algunos lenguajes modernos (como Lisp, Python o JavaScript) adoptan reglas mixtas de alcance y cómo afecta eso al razonamiento sobre el código?
Búsqueda: *lexical scope vs dynamic scope in Lisp, Python scope chain, closures in JavaScript*
Programa: Implementar un mini-lenguaje con dos modos de alcance seleccionables, ejecutando un mismo programa bajo ambos y comparando los resultados.

6. Estrategias de evaluación y optimización

Pregunta: ¿Cómo influyen las estrategias de evaluación (perezosa, ansiosa o diferida) en la eficiencia de un programa y su posibilidad de paralelización?
Búsqueda: *lazy evaluation performance, strictness analysis, parallel functional programming*
Programa: Implementar un generador de secuencias infinitas que calcule números primos o Fibonacci de manera perezosa y comparar rendimiento con la versión ansiosa.

7. Recursión, combinadores y autointerpretación

Pregunta: ¿Cómo puede utilizarse el combinador Y para construir programas que se autointerpretan o realizan meta-programación en un lenguaje funcional?
Búsqueda: *Y combinator meta-programming, self-interpreters in lambda calculus, fixed-point theory*
Programa: Implementar el combinador Y y usarlo para definir una función recursiva que interprete expresiones aritméticas simples.

8. Continuaciones y control del flujo

Pregunta: ¿Cómo puede el uso explícito de continuaciones modelar comportamientos como excepciones, backtracking o concurrencia cooperativa?
Búsqueda: *call/cc examples, continuation passing style, delimited continuations, control operators*
Programa: Implementar un programa que use continuaciones (let/cc) para simular manejo de errores o backtracking en una búsqueda.

9. Programas estructurados y optimización del control de flujo

Pregunta: ¿Cómo influyen las estructuras de control bien definidas en la verificabilidad y optimización automática de los programas?
Búsqueda: *structured control flow optimization, control-flow graph, static analysis structured programming*
Programa: Escribir un programa secuencial que genere una gráfica control de flujo de sí mismo y destaque las partes optimizables.

10. Recursión frente a iteración en la práctica de compiladores

Pregunta: ¿Qué implicaciones tiene elegir recursión o iteración en la generación de código máquina y en la optimización por el compilador?
Búsqueda: *tail call optimization, recursion vs loops compiler optimization, stack frames recursion*
Programa: Implementar versiones recursiva y iterativa de una misma función y medir su consumo de pila y tiempo en Haskell o Python.

11. Variables, referencias y estado mutable controlado

Pregunta: ¿Cómo puede introducirse mutabilidad controlada sin perder las garantías de pureza funcional?
Búsqueda: *monads for state, state monad Haskell, referential transparency with mutation*

Programa: Implementar un contador mutable usando la mónada State o una simulación explícita de memoria en un lenguaje puro.

12. Estrategias de paso de parámetros y evaluación diferida

Pregunta: ¿Cómo pueden combinarse estrategias de paso de parámetros (por valor, nombre y necesidad) para optimizar ejecución en lenguajes híbridos?
Búsqueda: *call-by-value vs call-by-need, hybrid evaluation strategies, parameter passing semantics*
Programa: Implementar un mini-intérprete que permita elegir la estrategia de evaluación de parámetros y mostrar las diferencias en ejecución.

13. Sistemas de tipos y verificación automática

Pregunta: ¿Cómo puede un sistema de tipos servir como primer nivel de verificación formal de propiedades en programas críticos?
Búsqueda: *type-based verification, refinement types, dependent types introduction, liquid Haskell*
Programa: Implementar un verificador de tipos que asegure que una división nunca divide entre cero.

14. Inferencia de tipos y detección de errores semánticos

Pregunta: ¿Cómo permite la inferencia de tipos detectar errores semánticos antes de la ejecución?
Búsqueda: *type inference, Hindley–Milner algorithm, semantic errors detection*
Programa: Implementar un algoritmo de inferencia simple que rechace expresiones donde se mezclan tipos incompatibles (por ejemplo, True + 3).

15. Polimorfismo y abstracción de datos

Pregunta: ¿Cómo puede el polimorfismo paramétrico combinarse con abstracción de datos para crear interfaces genéricas seguras?
Búsqueda: *parametric polymorphism, type abstraction, modules in functional languages*
Programa: Implementar una biblioteca genérica para listas, colas o árboles, con funciones polimórficas y tipos abstractos.

16. Inferencia de tipos y lógica de predicados

Pregunta: ¿Qué relación existe entre los sistemas de tipos y la inferencia lógica en sistemas formales de primer orden?
Búsqueda: *type inference and logic, Curry–Howard, type inference as proof search*
Programa: Implementar una versión mínima del algoritmo W y mostrar cómo cada paso puede verse como una inferencia lógica.

17. Propiedades de preservación y progreso en semántica operacional

Pregunta: ¿Cómo pueden demostrarse experimentalmente las propiedades de preservación y progreso mediante simulaciones computacionales?
Búsqueda: *type safety proofs, operational semantics simulation, type preservation check*
Programa: Implementar una máquina de evaluación que muestre cómo las expresiones bien tipadas nunca se bloquean.

18. Tipificado gradual y lenguajes híbridos

Pregunta: ¿De qué manera los lenguajes modernos combinan tipificado estático y dinámico (tipado gradual) para equilibrar flexibilidad y seguridad?
Búsqueda: *gradual typing, typed Racket, TypeScript, gradual type inference*
Programa: Implementar un mini-intérprete con tipificado opcional donde se permita mezclar expresiones tipadas y no tipadas.

19. Tipos dependientes y demostraciones computacionales

Pregunta: ¿Cómo pueden los tipos dependientes expresar y verificar propiedades matemáticas dentro del propio programa?
Búsqueda: *dependent types, proofs in Agda or Idris, total functions, proof-carrying code*
Programa: Implementar un ejemplo en Agda donde el tipo garantice que una función reverse conserva la longitud de la lista.

20. Seguridad de tipos y lenguajes para IA

Pregunta: ¿Cómo influyen los sistemas de tipos en el diseño de lenguajes seguros para Inteligencia Artificial y verificación de modelos?
Búsqueda: *type safety AI systems, probabilistic programming languages, typed neural networks, formal verification AI*
Programa: Implementar un mini-lenguaje que tipifique operaciones probabilísticas básicas (por ejemplo, muestreo o normalización de distribuciones).