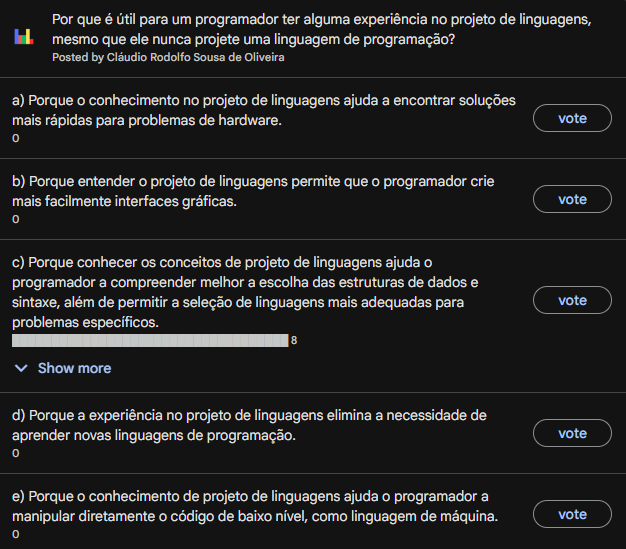
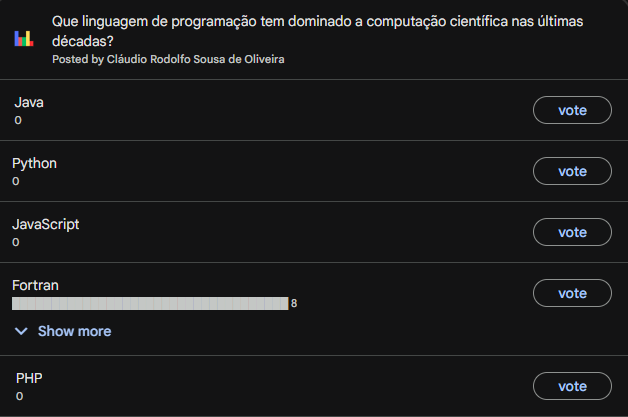
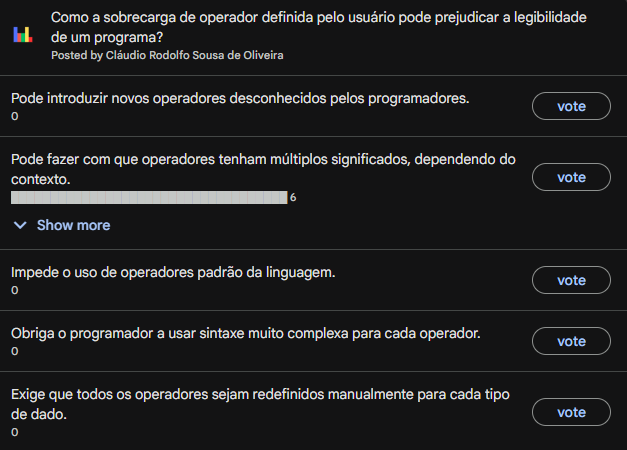
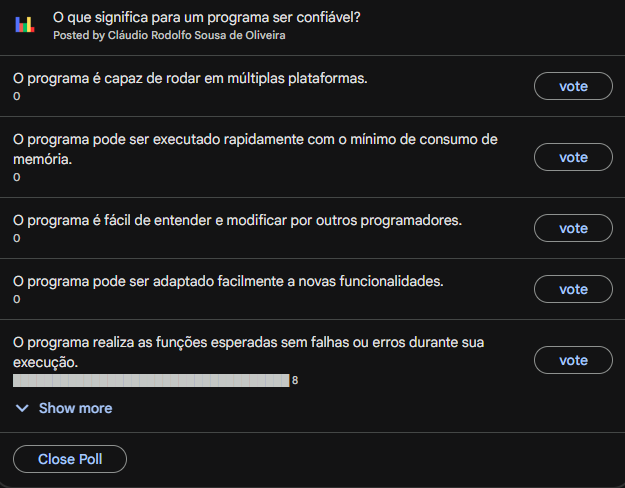
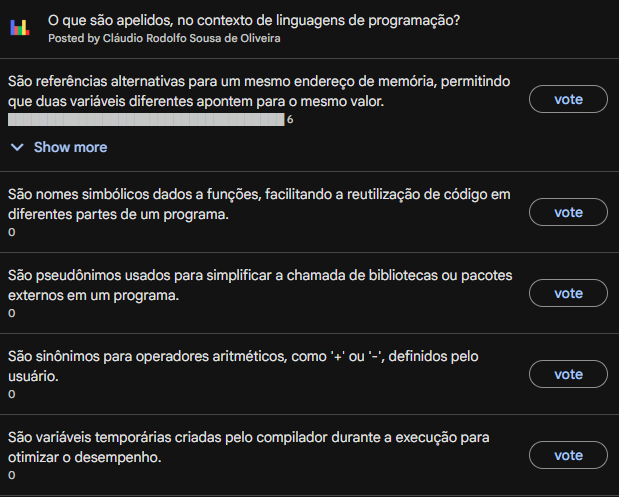
Todas as questões do google chat de PLP

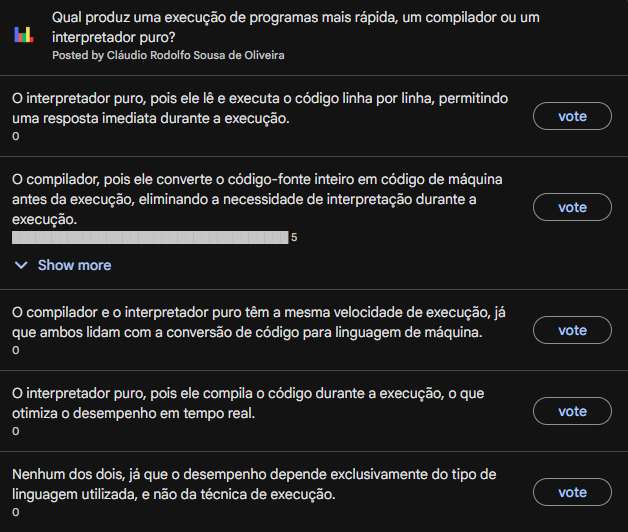


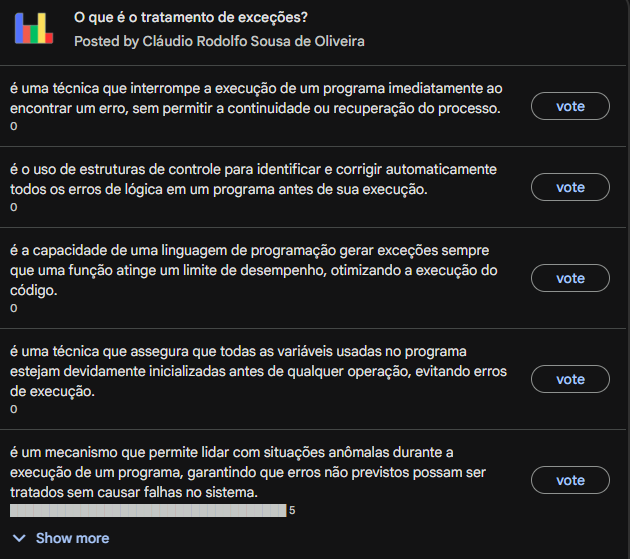


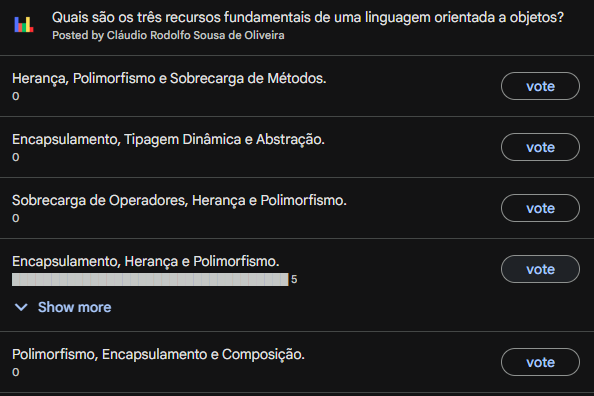


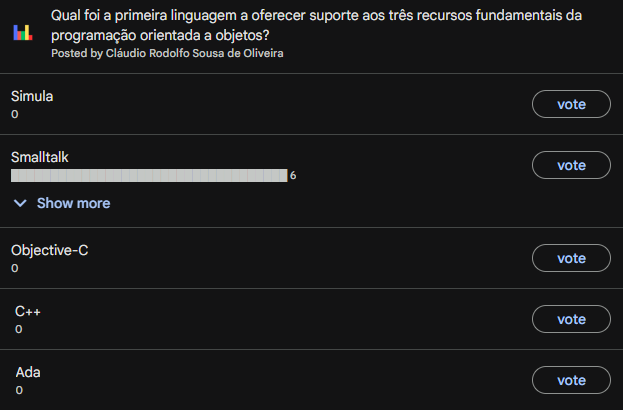


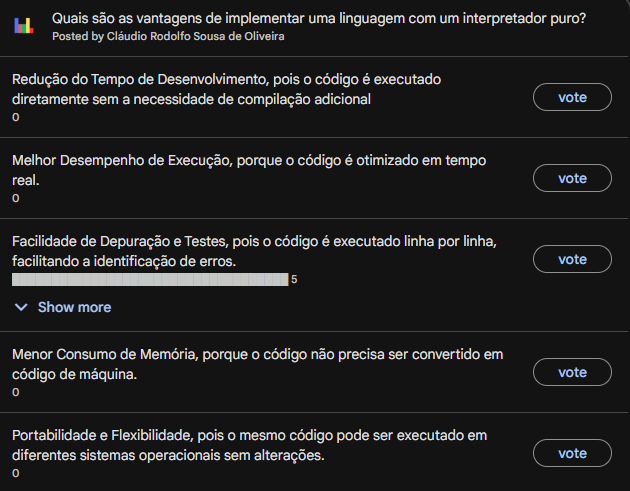


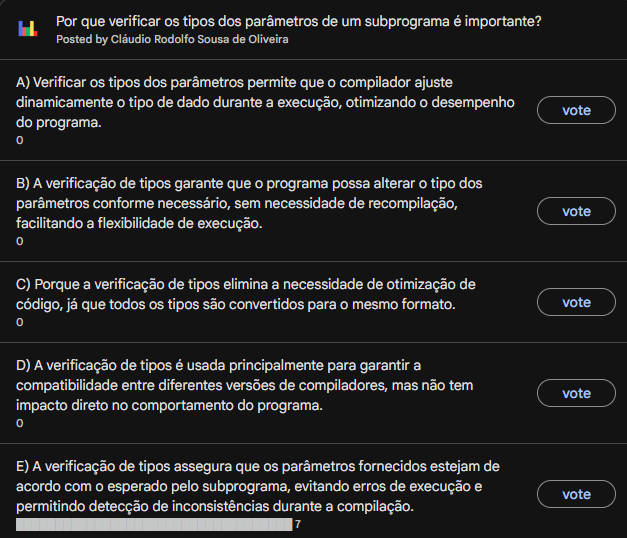


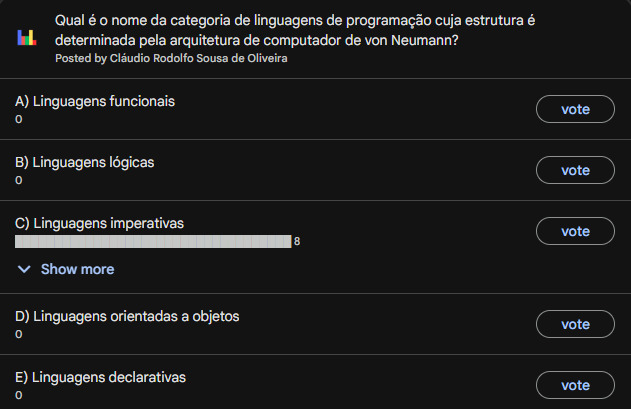


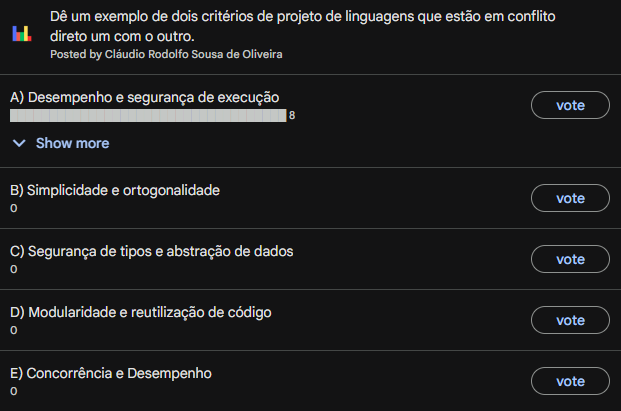


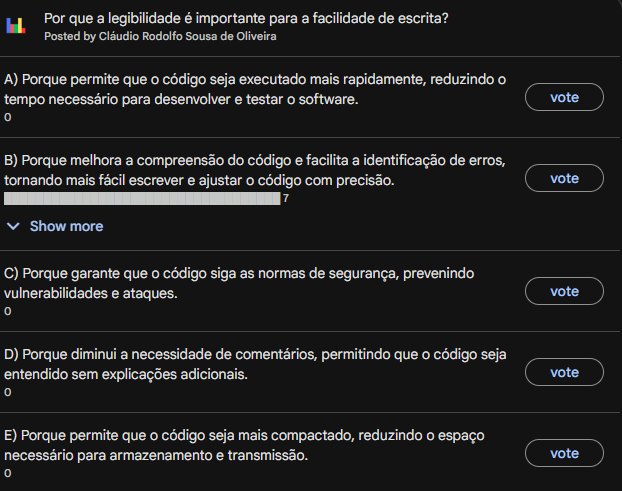


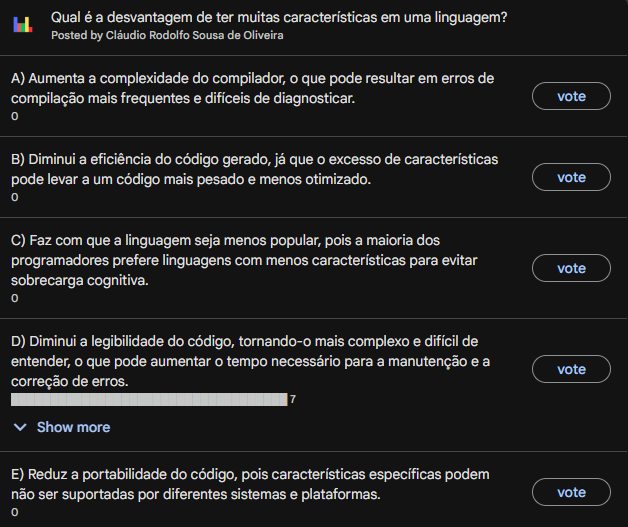




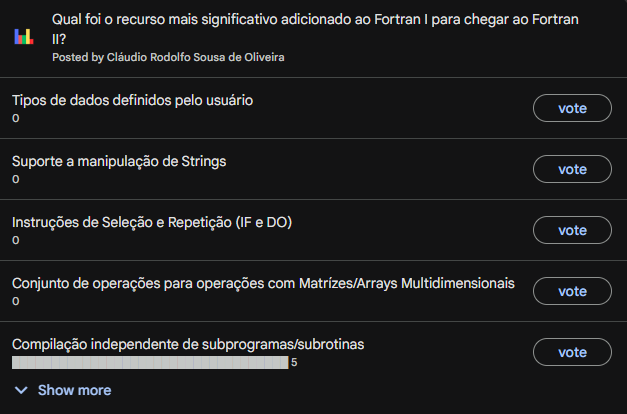


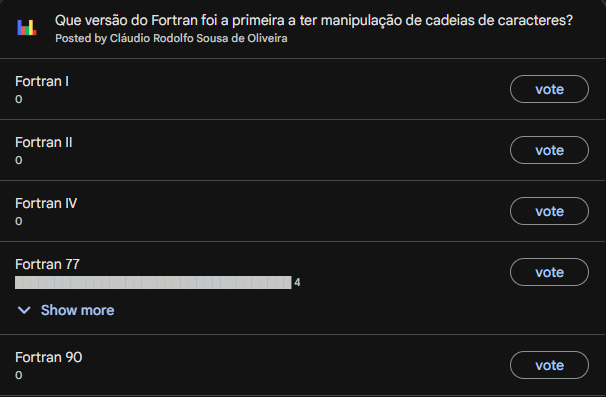




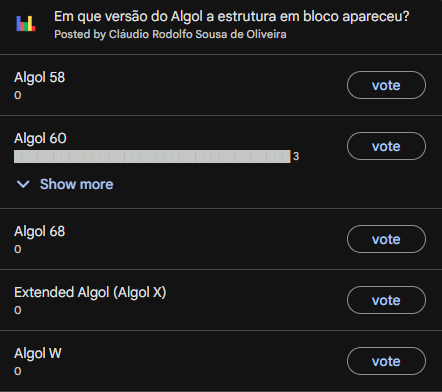




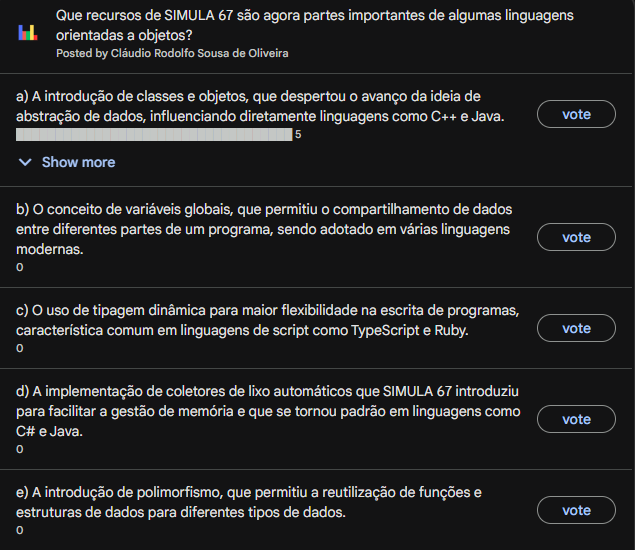


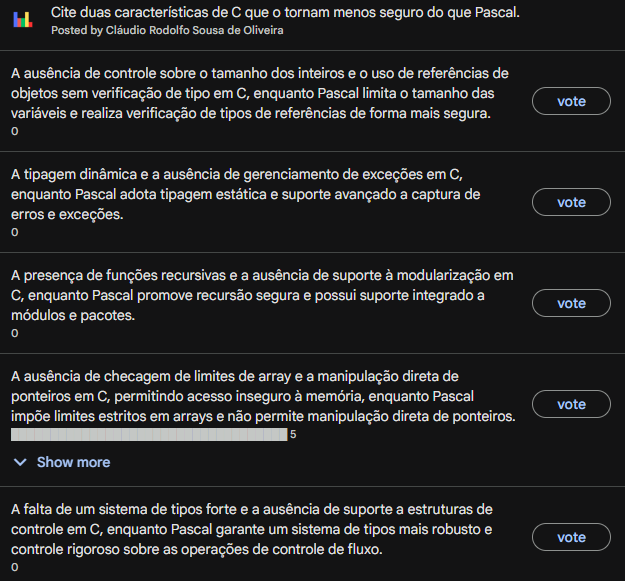


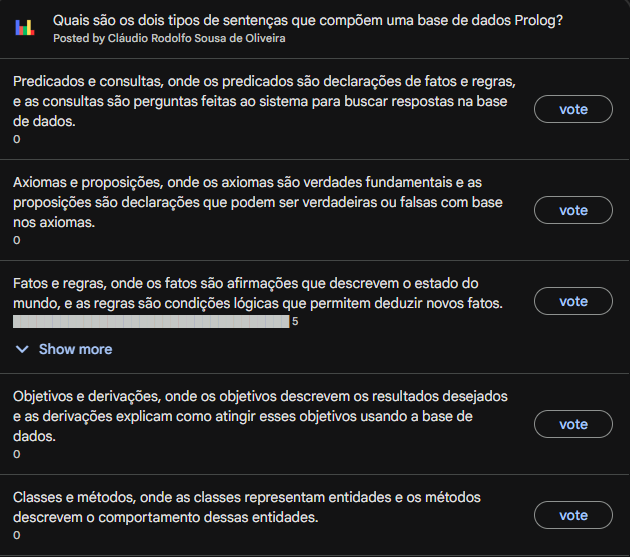


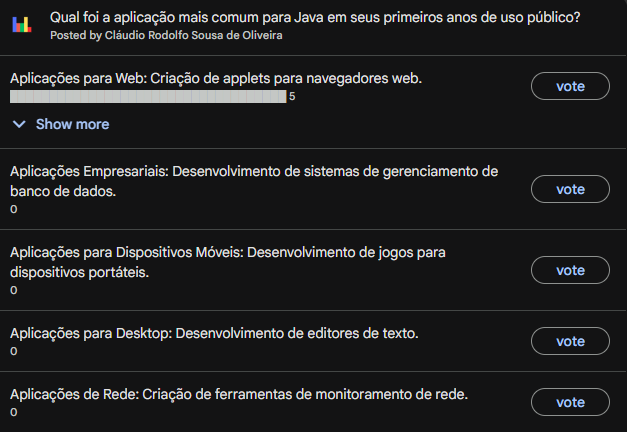






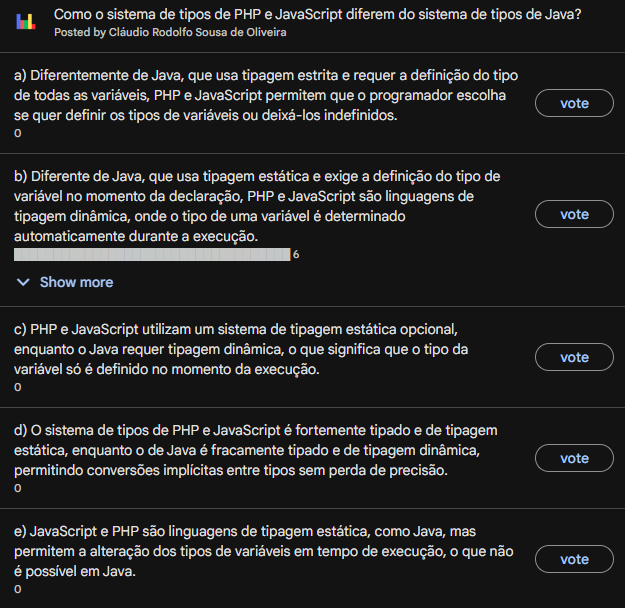


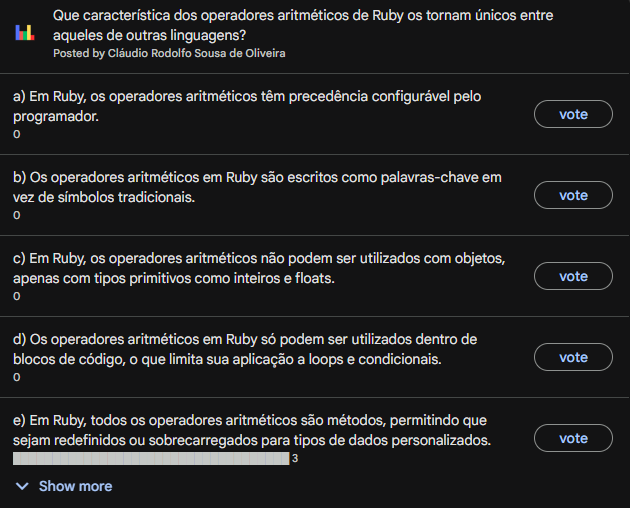


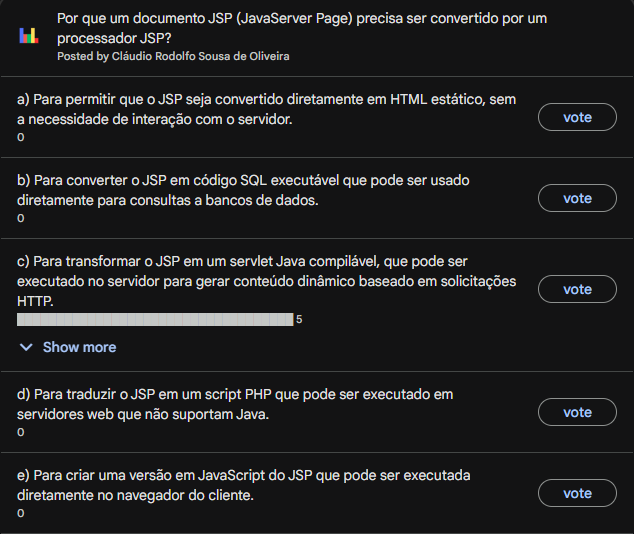


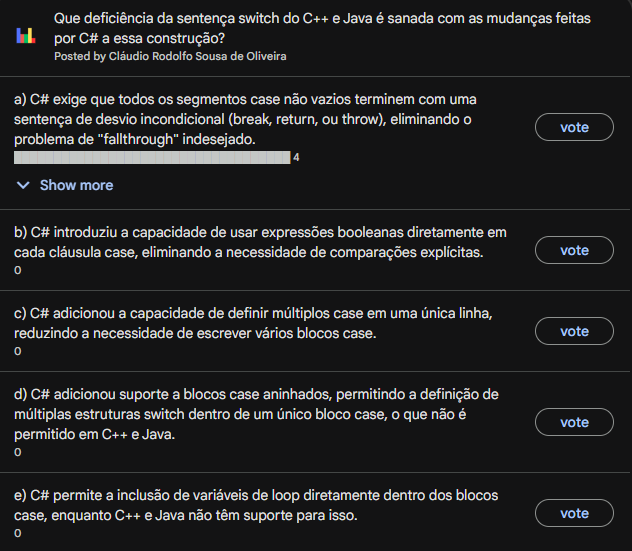
O Capítulo 2 do livro está disponível aqui: <https://classroom.google.com/c/Njg1MzUyMzM4OTE4/m/Njg1NDI1MzY1ODkx/details>  
  
Vou indicar as páginas do livro, com base neste arquivo, que vocês podem encontrar as respostas para as questões postadas.  
PS: Tem que ler o livro pessoal. 🙂  
  
Cite duas estruturas de dados comuns incluídas em Plankalkül.  
pág 59   
  
Qual foi o recurso mais significativo adicionado ao Fortran I para chegar ao Fortran II?  
pág 66  
  
Que versão do Fortran foi a primeira a ter manipulação de cadeias de caracteres?  
pág 66  
  
De que maneira as linguagens de programação Scheme e COMMON LISP são opostas?  
pág 72-73  
  
Em que versão do Algol a estrutura em bloco apareceu?  
pág 78

Que organização foi a maior responsável pelo sucesso inicial do COBOL (em termos de uso)?  
pág 82  
Que recursos de SIMULA 67 são agora partes importantes de algumas linguagens orientadas a objetos?  
pág 95  
Cite duas características de C que o tornam menos seguro do que Pascal.  
pág 101  
Quais são os dois tipos de sentenças que compõem uma base de dados Prolog?  
pág 102  
Qual foi a aplicação mais comum para Java em seus primeiros anos de uso público?  
pág 115

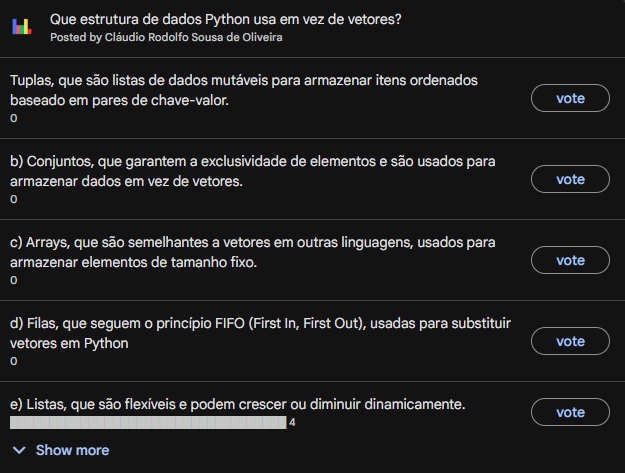








Como o sistema de tipos de PHP e JavaScript diferem do sistema de tipos de Java?  
pág 121-122  
  
Que estrutura de dados Python usa em vez de vetores?  
pág 123  
  
Que característica dos operadores aritméticos de Ruby os tornam únicos entre aqueles de outras linguagens?  
pág 123-124  
  
Que deficiência da sentença switch do C++ é sanada com as mudanças feitas por C# a essa construção?  
pág 126  
  
Por que um documento JSP (JavaServer Pages) precisa ser convertido por um processador JSP?  
pág 129



RESUMO DA EVOLUÇÃO DO FORTRAN (VERSÕES QUE O LIVRO TEXTO DO SEBESTA ABORDA)  
1. Fortran I (1957)  
Código compilado tão eficiente quanto código manual.  
estruturas de controle de fluxo IF aritmético (IF (expr) label1, label2, label3).  
Uso de loops DO, semelhantes aos atuais FOR (DO label var = start, end, step).  
Subrotinas/subprogramas.  
  
2. Fortran II (1958)  
Compilação de Subrotinas/subprogramas independentes.  
Compilador com link-editor.  
  
3. Fortran IV (1962)  
Declaração de tipos para variáveis.  
Construção lógica IF  
Subprogramas como parâmetros para outros subprogramas (SOMA(4,MAIOR(3,2)).  
  
4. Fortran 77  
Manipulação de cadeia de caracteres.  
Condições para laço.  
IF com ELSE.  
  
5. Fortran 90  
Funções nativas para manipulação de matrizes (Multiplicação, Transposição, etc.)  
Permite alocar dados dinamicamente (durante a  execução do programa).  
Ponteiros.  
Instrução Case, semelhante ao atual Switch..Case  
Instrução Exit, semelhante ao atual Break  
Intrução Cicle, semelhante ao atual Continue  
Subprogramas recursivos  
Criação de pacotes/módulos.  
Remoção de recursos de linguagens anteriores, como goto, if aritmético.

RESUMO DA EVOLUÇÃO DO ALGOL  
Algol 58  
Esforço internacional para criar uma linguagem para a expressão de algoritmos de maneira concisa.  
Serviu como base para a formalização da linguagem em versões futuras.  
Inspirou conceitos de definição de procedimentos e o uso de expressões aritméticas de forma clara.  
Apesar de ser uma versão preliminar, muitas das ideias do Algol 58 foram desenvolvidas e formalizadas nas versões seguintes.  
  
Algol 60  
Versão mais influente do Algol. É considerada uma das linguagens mais influentes da história da computação, especialmente por sua clareza e rigor na definição da sintaxe.  
Formalização por Backus-Naur Form (BNF): Usado para definir formalmente a gramática da linguagem, o que influenciou o design de compiladores e outras linguagens.  
Blocos de código: Introdução da estrutura em blocos, onde variáveis e sub-rotinas poderiam ser declaradas em escopos locais.  
Passagem de parâmetros por valor e por nome: Flexibilidade na forma como os argumentos eram passados para funções e procedimentos.  
Recursão: Algol 60 foi uma das primeiras linguagens a suportar recursão.  
Estrutura de controle clara: Introduziu comandos estruturados de controle, como if-then-else e loops como for.  
Inspirou o desenvolvimento de linguagens como Pascal, C, e Simula.  
  
Algol W (1966)  
Desenvolvido por Niklaus Wirth (criador do Pascal) como uma proposta de evolução para o Algol 60.  
Contribuições:  
Introduziu tipos de dados estruturados como registros (conhecidos como "structs" ou "records" em outras linguagens).  
A ênfase na simplicidade de Algol W inspirou diretamente o desenvolvimento da linguagem Pascal, que se tornaria uma linguagem amplamente usada em educação e desenvolvimento de sistemas.  
  
Algol 68  
Considerada uma versão mais complexa e poderosa do Algol 60.  
Introduziu a programação ortogonal (uma filosofia de design onde características da linguagem podiam ser combinadas de formas consistentes).  
Tipos de dados avançados: Suporte para arrays dinâmicos, strings e tipos de dados mais complexos, como uniões e estruturas.  
Tipos de dados fortes e verificação estática: Prevenção de erros ao garantir que as variáveis e os dados fossem verificados em tempo de compilação.  
Apesar de suas inovações, a complexidade de Algol 68 tornou difícil sua adoção em larga escala. No entanto, ele influenciou o design de muitas linguagens subsequentes, como Ada e C++.  
  
Algol X (Extended Algol)  
Um esforço para criar uma versão unificada e estendida do Algol, incorporando as lições aprendidas com o Algol 60 e o Algol 68, mas nunca foi amplamente adotada ou formalizada como uma linguagem oficial.

Uma das dúvidas da última aula, se Simula 67 tinha suporte aos 3 pilares da POO (Questão 23 do Capítulo 1)? Segundo o livro do Sebesta, não. A linguagem possui somente suporte a Abstração de Dados/Encapsulamento(por meio da definição de Classes e Objetos), não possuindo assim suporte a Herança e Polimorfismo, o que não deixa de ser um grande avanço. Por isso, não é considerada a primeira Linguagem Orientada a Objeto. Todos esses conceitos foram introduzidos pela primeira vez pela linguagem Smalltalk.

Qual a descrição EBNF para uma sentença de cabeçalho de definição de classe em Java?  
a) <definicao\_classe> ::= ( <modificador> )? "class" <identificador> ( "extends" <identificador> )? ( "implements" <identificador\_lista> )? (CERTA)  
b) <definicao\_classe> ::= ( <modificador> )? "class" <identificador> "(" <parametros> ")"      
c) <definicao\_classe> ::= <modificador> "class" <identificador> ( "implements" <identificador> )? ( "extends" <identificador\_lista> )?  
d) <definicao\_classe> ::= ( <modificador> )+ "class" <identificador>    
e) <definicao\_classe> ::= ( <modificador> )\* "class" <identificador> ( "extends" <identificador\_lista> )? ( "implements" <identificador\_lista> )?

Descreva, em português, a linguagem definida pela seguinte gramática:  
<S> ::= <A> <B> <C>  
<A> ::= "a" <A> | "a"  
<B> ::= "b" <B> | "b"  
<C> ::= "c" <C> | "c"  
  
a) A linguagem gerada contém cadeias que consistem em uma sequência de um número igual de "a"s, "b"s e "c"s, como "abc", "aabbcc", "aaabbbccc".  
b) A linguagem gerada contém cadeias que consistem em uma sequência de "a"s, seguida de "b"s, com um número fixo de "a"s e "b"s, terminando com exatamente três "c"s.  
c) A linguagem gerada contém cadeias que começam e terminam com qualquer número de "a"s e "b"s, seguidas de um ou mais "c"s.  
d) A linguagem gerada contém cadeias de números arbitrários de "a"s e "b"s, terminando com uma sequência de "c"s, onde a quantidade de "c"s deve ser igual à soma do número de "a"s e "b"s.  
e) A linguagem gerada contém cadeias que consistem em uma sequência de "a"s, seguida de "b"s, e depois "c"s, com pelo menos um símbolo de cada tipo. As quantidades de "a"s, "b"s e "c"s podem variar independentemente. (CERTA)

Considere a seguinte gramática:  
<S> ::= "a" <S> "c" <B> | <A> | "b"  
<A> ::= "c" <A> | "c"  
<B> ::= "d" | <A>  
Quais das sentenças abaixo estão na linguagem gerada por essa gramática?  
a) abbcd  
b) acccbd  
c) acccbcc  
d) acd  
e) accc (CERTA)

Que 3 extensões da BNF são comuns para a maioria das EBNFs?  
a) Múltipla escolha, Repetição, Opcional. (CERTA)  
b) Recursão, Condicional, encapsulamento.  
c) Operadores matemáticos, Laços, Funções.  
d) Atribuição, Exceção, Modularidade.  
e) Laços, Opções múltiplas, Escopo.

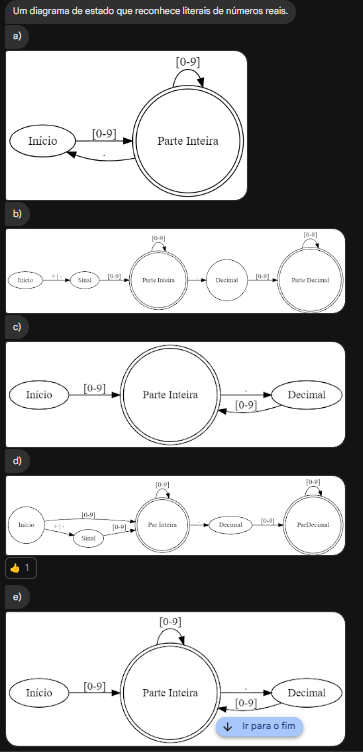
Defina lexema e token.  
a) Lexema é uma sequência de caracteres que representa uma expressão lógica, enquanto token é a conversão dessa expressão para um formato aritmético.  
b) Lexema é o nome dado à estrutura de controle que define o fluxo do programa, enquanto token é a representação numérica de variáveis globais e locais.  
c) Lexema é a descrição formal das regras sintáticas de uma linguagem de programação, enquanto token refere-se a uma operação semântica específica realizada durante a execução do código.  
d) Lexema é a menor unidade de significado em uma linguagem de programação, enquanto token é uma categoria abstrata associada a um lexema, indicando seu tipo semântico, como palavra-chave, operador ou identificador. (CERTA)  
e) Lexema refere-se ao conteúdo de uma variável armazenada em um registrador, enquanto token é a instrução que define o escopo de memória do código.

usando a gramática do Exemplo 3.2 mostre uma derivação à extrema esquerda da seguinte instrução:  
A=A\*(B+(C))  
  
//Gramática do Exemplo 3.2  
<atribuicao> ::= <id> "=" <expr>  
<id> ::= "A" | "B" | "C"  
<expr> ::= <id> "+" <expr> | <id> "\*" <expr> | "(" <expr> ")" | <id>  
  
a)   
<atribuicao>   
<id> = <expr>  
<id> = <id> \* <expr>  
<id> = <id> \* (<expr>)  
<id> = <id> \* (<id> + <expr>)  
<id> = <id> \* (<id> + (<expr>))  
<id> = <id> \* (<id> + (<id>))  
A = A \* (B + (C))  
  
b)   
A = A \* (B + (C))  
<id> = <id> \* (<id> + (<id>))  
<id> = <id> \* (<id> + (<expr>))  
<id> = <id> \* (<id> + <expr>)  
<id> = <id> \* (<expr>)  
<id> = <id> \* <expr>  
<id> = <expr>  
<atribuicao>   
  
c) (CERTA)  
<atribuicao>   
<id> = <expr>  
A = <expr>  
A = <id> \* <expr>  
A = A \* <expr>  
A = A \* (<expr>)  
A = A \* (<id> + <expr>)  
A = A \* (B + <expr>)  
A = A \* (B + (<expr>))  
A = A \* (B + (<id>))  
A = A \* (B + (C))  
  
d)   
A = A \* (B + (C))  
A = A \* (B + (<id>))  
A = A \* (B + (<expr>))  
A = A \* (B + <expr>)  
A = A \* (<id> + <expr>)  
A = A \* (<expr>)  
A = A \* <expr>  
A = <id> \* <expr>  
A = <expr>  
<id> = <expr>  
<atribuicao>   
  
e)   
<atribuicao>   
<id> = <expr>  
A = <expr>  
A = <id> \* <expr>  
A = A \* <expr>  
A = A \* (<expr>)  
A = A \* (<id> + <expr>)  
A = A \* (B + <expr>)  
A = A \* (B + (<expr>))  
A = A \* (B + (<id> \* <expr>))  
A = A \* (B + (<id>))  
A = A \* (B + (C))

Descreva semântica estática e dinâmica.  
a) Semântica estática refere-se às regras de execução do programa, como controle de fluxo e manipulação de memória. Semântica dinâmica lida com a verificação de tipos e variáveis antes do programa ser compilado.  
b) Semântica estática é a verificação das regras que podem ser analisadas antes da execução, como a verificação de tipos e escopo durante a compilação. Semântica dinâmica refere-se ao comportamento durante a execução do programa, como a manipulação de memória e o tratamento de exceções. (CERTA)  
c) Semântica estática lida com a execução de operações aritméticas e de controle de fluxo em tempo de execução. Semântica dinâmica trata de garantir que variáveis e funções estejam no escopo correto antes da execução do programa.  
d) Semântica estática lida com a otimização do código durante o processo de execução do programa. Semântica dinâmica envolve a análise sintática e a interpretação de instruções do programa antes de ele ser executado.  
e) Semântica estática é responsável pela análise léxica e sintática do código durante a compilação. Semântica dinâmica trata da verificação de tipos e escopo antes do programa ser compilado.

Defina sintaxe e semântica.  
a) Sintaxe especifica as regras que descrevem como o código deve ser escrito corretamente. Semântica refere-se ao significado do código, ou seja, o que ele faz. (CERTA)  
b) Sintaxe refere-se ao comportamento do programa durante a execução. Semântica define as regras formais de escrita de um código.  
c) Sintaxe descreve o comportamento dinâmico do código em tempo de execução. Semântica trata da verificação de erros sintáticos durante a compilação.  
d) Sintaxe define as verificações de tipos e variáveis antes da execução do programa. Semântica define as regras formais para a escrita do código.  
e) Sintaxe descreve o significado das instruções em um programa. Semântica define a ordem de execução das instruções, garantindo que elas sigam a lógica do programa.

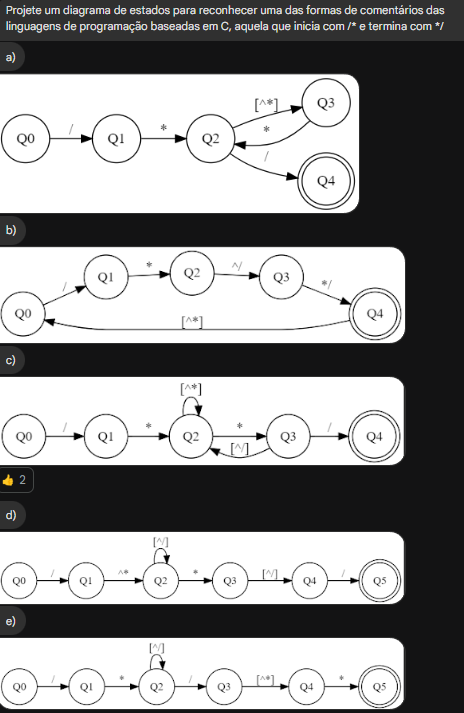
Quais as principais tarefas de uma analisador léxico?  
a) O analisador léxico gera a árvore sintática do programa, organizando os tokens de acordo com a gramática da linguagem.  
b) O analisador léxico compila o código-fonte em código de máquina e garante que todas as funções estejam corretamente implementadas.  
c) O analisador léxico é responsável por traduzir o código-fonte em linguagem de máquina e otimizar a execução do programa.  
d) O analisador léxico verifica o fluxo de controle do programa, como loops e condicionais, antes de enviar o código para a execução.  
e) O analisador léxico lê o código-fonte, divide-o em tokens, ignora espaços em branco e comentários, e reporta erros léxicos quando necessário.(CERTA)



Qual a principal diferença entre um atributo sintetizado e um atributo herdado?  
a) Um atributo sintetizado depende dos valores dos irmãos de um nó na árvore sintática, enquanto um atributo herdado é calculado com base nos filhos desse nó.  
b) Um atributo sintetizado propaga informações semânticas de baixo para cima na árvore, enquanto um atributo herdado propaga informações semânticas de cima para baixo. (CERTA)  
c) Um atributo sintetizado sempre utiliza valores constantes definidos no início do programa, enquanto um atributo herdado é calculado dinamicamente durante a execução.  
d) Um atributo sintetizado é responsável pela análise semântica de uma linguagem, enquanto um atributo herdado cuida exclusivamente da análise sintática.  
e) Um atributo sintetizado ocorre apenas em linguagens com tipagem dinâmica, enquanto um atributo herdado é específico de linguagens com tipagem estática.

A qual propósito servem os predicados em uma gramática de atributos?  
a) Os predicados garantem que os atributos calculados em uma gramática de atributos estejam corretos, verificando condições semânticas que devem ser satisfeitas.(CERTA)  
b) Predicados nas gramáticas de atributos são responsáveis por determinar a precedência dos operadores durante a análise sintática de expressões aritméticas.  
c) Em gramáticas de atributos, os predicados são usados para otimizar o tempo de execução de programas, garantindo que não haja cálculos desnecessários durante a execução.  
d) Os predicados em gramáticas de atributos determinam a ordem de avaliação dos atributos durante a execução do programa, organizando a sequência em que eles são processados.  
e) Predicados são usados para selecionar o tipo correto de dado em tempo de execução, alterando dinamicamente o comportamento do código baseado no valor dos atributos.

Qual a diferença entre um atributo sintetizado e um atributo intrínseco?  
a) Atributos sintetizados são aqueles que precisam ser fornecidos explicitamente no código pelo programador, enquanto atributos intrínsecos são calculados automaticamente pelo compilador com base no contexto.  
b) Atributos sintetizados são usados para verificar a correção semântica do programa, enquanto atributos intrínsecos são usados apenas para verificar a sintaxe do código.  
c) Um atributo sintetizado pode ser calculado usando a informação de qualquer nó da árvore sintática, enquanto um atributo intrínseco é sempre herdado de nós pais na árvore sintática.  
d) Atributos sintetizados dependem de regras semânticas globais e são aplicáveis em qualquer nível da árvore sintática, enquanto atributos intrínsecos são associados a regras sintáticas locais.  
e) Atributos sintetizados são calculados a partir dos atributos dos filhos de um nó na árvore sintática, enquanto um atributo intrínseco é diretamente associado à estrutura da linguagem e não depende de nenhum outro nó.(CERTA)



Escreva uma gramática de atributo cuja base da BNF é aquela do Exemplo 3.6 na Seção 3.4.5, mas cujas regras da linguagem são as seguintes: os tipos de dados não podem ser misturados em expressões, mas as sentenças de atribuição não precisam ter os mesmos tipos em ambos os lados do operador de atribuição.  
BNF do Exemplo 3.6  
<atribuicao> → <var> = <expr>  
<expr> → <var> + <var> | <var>  
<var> → A | B | C  
  
a)  
<atribuicao> → <var> = <expr>  
<expr> → <var> + <var>'  
    Regras Semânticas:  
        <expr>.iguais\_lado\_direito ⭠ if ( <var>.tipo\_atual = <var>'.tipo\_atual) then false else true endif  
    Predicado:  
        <expr>.iguais\_lado\_direito  
<expr> → <var>  
    Regras Semânticas:  
        <expr>.iguais\_lado\_direito ⭠ false  
<var> → A | B | C  
    Regras Semânticas:  
        <var>.tipo\_atual ⭠ lookup(<var>.token)  
  
b)  
<atribuicao> → <var> = <expr>  
<expr> → <var> + <var>'  
    Regras Semânticas:  
        <expr>.tipo\_atual ⭠ if ( <var>.tipo\_atual = <var>'.tipo\_atual) then num\_real else num\_int endif  
    Predicado:  
        <expr>.tipo\_atual = <expr>.tipo\_esperado  
<expr> → <var>  
    Regras Semânticas:  
        <expr>.tipo\_atual ⭠ num\_int  
<var> → A | B | C  
    Regras Semânticas:  
        <var>.tipo\_atual ⭠ lookup(<var>.token)  
  
c)  
<atribuicao> → <var> = <expr>  
<expr> → <var> + <var>'  
    Regras Semânticas:  
        <expr>.iguais\_lado\_direito ⭠ if ( <var>.tipo\_atual = <var>'.tipo\_atual) then false else true endif  
    Predicado:  
        <expr>.iguais\_lado\_direito  
<expr> → <var>  
    Regras Semânticas:  
        <expr>.iguais\_lado\_direito ⭠ <var>.tipo\_atual  
<var> → A | B | C  
    Regras Semânticas:  
        <var>.tipo\_atual ⭠ lookup(<var>.token)  
  
d) (CERTA)  
<atribuicao> → <var> = <expr>  
<expr> → <var> + <var>'  
    Regras Semânticas:  
        <expr>.iguais\_lado\_direito ⭠ if ( <var>.tipo\_atual = <var>'.tipo\_atual) then true else false endif  
    Predicado:  
        <expr>.iguais\_lado\_direito  
<expr> → <var>  
    Regras Semânticas:  
        <expr>.iguais\_lado\_direito ⭠ true  
<var> → A | B | C  
    Regras Semânticas:  
        <var>.tipo\_atual ⭠ lookup(<var>.token)  
  
e)  
<atribuicao> → <var> = <expr>  
<expr> → <var> + <var>'  
    Regras Semânticas:  
        <expr>.tipo\_esperado ⭠ if ( <var>.tipo\_atual = <var>'.tipo\_atual) then num\_int else num\_real endif  
    Predicado:  
        <expr>.tipo\_esperado = <var>.tipo\_atual  
<expr> → <var>  
    Regras Semânticas:  
        <expr>.tipo\_esperado ⭠ lookup(<var>.tipo\_atual)  
<var> → A | B | C  
    Regras Semânticas:  
        <var>.tipo\_atual ⭠ <var>.token

De que maneira as palavras reservadas são melhores do que as palavras-chave?   
a) Palavras reservadas garantem que certos identificadores tenham significados fixos e não possam ser usados para outros fins, como nomes de variáveis ou funções, o que evita ambiguidades no código. (CERTA)  
b) Palavras reservadas são usadas principalmente em linguagens compiladas, o que as torna mais eficientes do que palavras-chave, que são mais comuns em linguagens interpretadas, onde o código é processado de maneira mais lenta.  
c) Ao contrário das palavras-chave, palavras reservadas podem ser redefinidas ou reutilizadas como identificadores, o que aumenta a flexibilidade do desenvolvedor durante a codificação.  
d) Palavras reservadas são mais curtas e fáceis de serem lembradas, facilitando a escrita do código, ao contrário das palavras-chave, que são mais longas e difíceis de memorizar.  
e) Palavras reservadas são melhores quando se trabalha com otimizações de compiladores, pois permitem que o compilador faça otimizações mais eficazes em tempo de compilação, diferentemente das palavras-chave, que são apenas interpretadas como comandos diretos.

Defina escopo estático e escopo dinâmico.  
a) Escopo estático refere-se à definição das variáveis em tempo de execução, onde a variável mais recentemente declarada é utilizada em qualquer ponto do código, enquanto escopo dinâmico depende da estrutura do código-fonte e é determinado em tempo de compilação.  
b) Escopo estático é quando as variáveis são alocadas em memória em tempo de execução e permanecem acessíveis durante todo o tempo de execução do programa. Escopo dinâmico ocorre quando as variáveis são alocadas em tempo de compilação e só estão disponíveis durante o tempo de execução da função que as declara.  
c) Escopo estático define a visibilidade das variáveis em tempo de compilação, pela organização do código. No escopo dinâmico, a visibilidade das variáveis é determinada em tempo de execução, de acordo com a sequência das chamadas de função. (CERTA)  
d) No escopo estático, a visibilidade de variáveis é baseada no ciclo de vida de um programa, onde o compilador gerencia as variáveis globalmente. Já no escopo dinâmico, as variáveis são liberadas assim que a execução de uma função termina, independentemente da organização do código.  
e) Escopo estático significa que todas as variáveis declaradas no início do programa podem ser acessadas de qualquer ponto do código, enquanto escopo dinâmico restringe o uso de variáveis ao seu contexto local, dependendo da função em que estão declaradas.

Depois do projeto e da implementação da linguagem, quais são os 4 momentos em que vinculações podem desenvolver-se em um programa?  
a) Tempo de configuração do ambiente de desenvolvimento, tempo de análise léxica, tempo de execução de testes unitários e tempo de compilação de funções auxiliares.  
b) Tempo de execução, tempo de depuração, tempo de validação e tempo de teste do sistema.  
c) Na fase de construção do código, tempo de montagem de bibliotecas, tempo de carregamento de módulos externos e na etapa de interpretação.  
d) Tempo de inicialização do sistema, a fase de construção do executável, a vinculação em tempo real durante a alocação de memória e tempo de compilação de estruturas de dados.  
e) Tempo de execução, tempo de compilação, tempo de carregamento e tempo de link-edição. (CERTA)

Quais as vantagens das constantes nomeadas?  
a) As constantes nomeadas são sempre mais rápidas em termos de desempenho do que as variáveis, pois o compilador pode otimizar seu uso durante a execução, melhorando a performance geral do programa.  
b) Utilizar constantes nomeadas reduz a quantidade de código que deve ser escrito, pois elas permitem que o programador elimine a necessidade de repetir valores em diferentes partes do código, resultando em um código mais curto e menos organizado.  
e) As constantes nomeadas não podem ser alteradas em tempo de execução, permitindo que os programadores adaptem seus valores conforme necessário, oferecendo maior flexibilidade em comparação com variáveis convencionais.  
d) As constantes nomeadas tornam o código mais legível, pois permitem que os programadores compreendam rapidamente o significado de um valor sem precisar decifrar valores literais, melhorando a manutenção e a compreensão do código. (CERTA)  
e) Utilizar constantes nomeadas ajuda a evitar erros de codificação, pois, ao invés de inserir o mesmo valor repetidamente, o programador utiliza um único identificador que representa aquele valor, garantindo que ele seja consistente em todo o código.

Quais as vantagens da vinculação dinâmica de tipos?  
a) A vinculação dinâmica de tipos elimina a necessidade de verificação de tipos em tempo de compilação, resultando em um código mais leve e mais fácil de depurar, diminuindo o tempo total de desenvolvimento.  
b) A vinculação dinâmica de tipos permite que variáveis sejam atribuídas a diferentes tipos de dados durante a execução do programa, o que aumenta a flexibilidade na manipulação de dados e na implementação de algoritmos. (CERTA)  
c) A vinculação dinâmica de tipos promove um maior controle sobre os tipos de dados utilizados em um programa, permitindo que os desenvolvedores capturem e tratem exceções de tipo em tempo de execução, garantindo assim que os tipos sejam consistentes.  
d) A vinculação dinâmica de tipos facilita a prototipagem rápida de aplicativos, permitindo que os desenvolvedores criem e testem ideias rapidamente sem se preocupar com a definição de tipos rigorosos, embora isso possa levar a erros em produção.  
e) A vinculação dinâmica de tipos pode resultar em um código mais difícil de manter, pois a natureza flexível dos tipos pode levar a eliminação de erros que são difíceis de detectar antes da execução.

Quais são as questões de projeto referentes a nomes?  
a) Devem ser considerados o comprimento máximo permitido dos nomes, a distinção entre letras maiúsculas e minúsculas, a presença de caracteres especiais e se palavras chaves podem ser usadas como nomes. (CERTA)  
b) As principais questões de projeto são determinar se os nomes de variáveis podem ser compartilhados entre diferentes blocos de código e se é permitido o uso de números como o primeiro caractere em um nome de variável.  
c) O foco principal do projeto de nomes de variáveis é garantir que os nomes sejam únicos em todo o código, independentemente do escopo, auxiliando assim a analise de regras de escopo e reutilização de palavras.  
d) A consideração mais relevante sobre nomes de variáveis é se eles são compostos por letras e números. A necessidade de verificar limitações sobre a reutilização de palavras reservadas ou a distinção de letras maiúsculas não é relevante.  
e) A questão fundamental é garantir que os nomes de variáveis sejam sempre autoexplicativos e semanticamente ricos, facilitando o entendimento do código por outros programadores, independentemente de regras de escopo ou reutilização de identificadores.

Defina vinculação estática e vinculação dinâmica.   
a) Vinculação dinâmica permite que variáveis mudem seu tipo em tempo de execução, enquanto a vinculação dinâmica fixa, também chamada de estática, o tipo de uma variável assim que ela é declarada, limitando sua flexibilidade ao longo da execução do programa.  
b) Vinculação estática ocorre em tempo de compilação, onde as associações entre variáveis e seus tipos, valores ou funções, por exemplo, são definidas e fixas. Vinculação dinâmica ocorre em tempo de execução, onde essas associações são determinadas conforme o programa é executado.(CERTA)  
c) Vinculação estática é utilizada principalmente em linguagens interpretadas, onde o tipo de dado é definido durante a execução do programa. Vinculação dinâmica, ao contrário, é característica das linguagens compiladas, onde o tipo de dado é determinado em tempo de compilação.  
d) Vinculação estática e vinculação dinâmica são termos intercambiáveis em linguagens modernas, já que ambas se referem ao processo de determinar como as variáveis são armazenadas na memória, sem diferenças no momento em que isso ocorre.  
e) Vinculação estática refere-se ao momento em que uma variável é associada a seu valor inicial no momento de sua declaração, enquanto vinculação dinâmica permite que essa associação seja alterada quando novas instruções são encontradas durante o fluxo de execução.

Qual é o problema geral com o escopo estático?  
a) O escopo estático cria ambiguidade na resolução de variáveis, pois a execução do programa pode alterar dinamicamente o escopo de uma variável, dependendo do caminho que a execução tomar, o que causa problemas na previsibilidade do código.  
b) O escopo estático depende da ordem em que as funções são chamadas em tempo de execução, o que torna o comportamento do programa imprevisível, especialmente quando funções recursivas estão envolvidas.  
c) O maior problema com o escopo estático é que ele pode levar a conflitos de nomes entre variáveis globais e locais, o que causa erros de execução, já que as variáveis globais têm precedência no escopo estático.  
d) O problema geral com o escopo estático é que ele pode tornar difícil para os desenvolvedores prever como as variáveis serão acessadas em programas grandes e com múltiplos blocos aninhados, especialmente se o código for altamente modular ou distribuído em várias bibliotecas. (CERTA)  
e) Escopo estático é problemático em linguagens que não suportam funções aninhadas, já que isso limita a capacidade de definir variáveis locais em escopos mais profundos e acessá-las de fora desses escopos durante a execução do programa.

Quais as vantagens das declarações implícitas de variáveis?  
a) Com declarações implícitas, as variáveis podem ser declaradas sem inicialização, e o compilador atribui automaticamente um valor padrão, o que facilita a manutenção de código complexo.  
b) Declarações implícitas permitem que o tipo da variável seja alterado dinamicamente durante a execução do programa, sem a necessidade de redefinição, proporcionando mais flexibilidade..  
c) O uso de declarações implícitas assegura que as variáveis sejam convertidas automaticamente entre tipos diferentes durante a execução, eliminando a necessidade de coerção de tipos por parte do desenvolvedor.  
d) As declarações implícitas de variáveis permitem a dedução automática de tipos pelo compilador com base nos valores atribuídos, economizando tempo de desenvolvimento, sem comprometer a verificação de tipos em tempo de compilação.(CERTA)  
e) Declarações implícitas eliminam a necessidade de inicializar variáveis antes de usá-las, permitindo que o compilador automaticamente forneça valores iniciais com base no tipo de dado da variável, economizando tempo de desenvolvimento.

O que é um bloco de código?

a) Um bloco de código é uma seção de um programa onde todas as variáveis declaradas dentro dele automaticamente se tornam variáveis globais, permitindo seu uso em qualquer parte do programa, independentemente do local de sua declaração.

b) Um bloco de código é uma estrutura que define um grupo de instruções que sempre compartilha o mesmo escopo com as variáveis de sua função externa, tornando possível acessar e modificar essas variáveis diretamente a partir do bloco.

c) Um bloco de código é uma seção delimitada por chaves ou indentação que define um novo escopo, onde variáveis declaradas dentro dele são locais e não são acessíveis fora do bloco, o que ajuda a organizar o código e evitar conflitos de variáveis.(CERTA)

d) Um bloco de código é um conjunto de expressões lógicas que determinam o fluxo do programa, mas não possuem um escopo próprio, fazendo com que as variáveis definidas nele estejam acessíveis em qualquer parte do programa.

e) Um bloco de código é uma região que compartilha o mesmo escopo de execução com todas as funções do programa, o que garante que variáveis declaradas em um bloco possam ser acessadas em qualquer outro bloco sem restrições.

Como um valor numérico real do tipo decimal (ponto fixo) perde espaço em memória?  
a) Ao utilizar mais bits para a parte inteira do que para a parte fracionária, os números que exigem precisão decimal podem não ser representados com exatidão, resultando em espaço perdido devido à necessidade de arredondamento.  
b) Valores decimais perdem espaço em memória quando a representação é feita usando um número fixo de bits para a parte fracionária, independentemente do valor a ser armazenado, fazendo com que pequenos números consumam o mesmo espaço que números grandes.(CERTA)  
c) A representação de valores decimais aumenta a complexidade do algoritmo, exigindo mais bits para operações aritméticas, o que resulta em perda de espaço em memória.  
d) Ao representar números decimais, pode haver perda de espaço se a quantidade de bits usados para a parte inteira for insuficiente para acomodar valores grandes, forçando a utilização de representações alternativas que consomem mais espaço.  
e) Valores decimais perdem espaço em memória devido à limitação de representar apenas valores em um intervalo específico, resultando em desperdício de espaço quando os valores estão muito abaixo ou acima do intervalo representável.

Quais são as questões de projeto mais importantes para tipos de cadeias de caracteres?  
a) As cadeias de caracteres devem armazenadas em memória contígua para maximizar o desempenho. Garantir que elas possam ser convertidas diretamente para outros tipos de dados.  
b) As cadeias de caracteres devem ser tratadas como coleções de variáveis inteiras. Garantir a compatibilidade entre sistemas operacionais diferentes.  
c) As cadeias de caracteres devem ser projetadas com base em um modelo de árvore ou listas ligadas. Permitir tanto tamanhos fixos quanto variáveis.  
d) As cadeias de caracteres devem ser um tipo especial de vetor de caracteres ou um tipo primitivo. Devem ter tamanho estático ou dinâmico. (CERTA)  
e) As cadeias de caracteres devem ser projetadas como objetos mutáveis ou imutáveis. Devem ser implementadas como um tipo de dado numérico especial.

Quais são as vantagens dos tipos de enumeração definidos pelo usuário?  
a) Permitem associar automaticamente valores numéricos a cada item, facilitando o uso de tipos de enumeração como índices em arrays, garantindo uma execução mais eficiente.  
b) A principal vantagem é que os valores de enumeração podem ser modificados em tempo de execução, permitindo maior flexibilidade para ajustar o comportamento do programa conforme necessário.  
c) Os tipos de enumeração definidos pelo usuário eliminam a necessidade de conversão entre diferentes tipos de dados, permitindo que uma variável enum seja automaticamente convertida em strings quando usados em operações de entrada e saída, o que facilita o processo de depuração do código.  
d) Oferecem a capacidade de definir valores de enumeração não duplicados para o mesmo nome de enum, permitindo uma maior reutilização de identificadores sem problemas de escopo.  
e) Permitem que o programador defina um conjunto restrito de valores que uma variável pode assumir, melhorando a legibilidade do código e a segurança, já que previnem o uso de valores fora do conjunto definido.(CERTA)

Quais são as questões de projeto para uniões?  
a) Se as uniões devem exigir a verificação de tipos e se podem ser usadas dentro de registros?(CERTA)  
b) Se as uniões devem permitir armazenar múltiplos valores ao mesmo tempo e se devem otimizar o uso de memória em sistemas que lidam com grandes quantidades de dados.  
c) Se a união deve ser usada apenas para tipos numéricos ou se devem evitar que o programador utilize tipos de dados complexos como strings ou estruturas em uniões.  
d) Se as uniões devem ser mutáveis ou se podem permitir que seu conteúdo seja definido apenas uma vez e não alterado posteriormente, melhorando a segurança no acesso à memória.  
e) Se as uniões devem implementar polimorfismo, ou se devem não permitir que diferentes métodos sejam chamados dependendo do tipo de dado atualmente armazenado na união.

Como JavaScript suporta matrizes esparsas?  
a) JavaScript lida com arrays esparsos atribuindo a cada índice um valor null como padrão para posições vazias, o que otimiza o armazenamento de memória e permite que o array seja percorrido sem erros.  
b) JavaScript cria uma estrutura de árvore internamente para armazenar matrizes esparsas, garantindo que cada elemento seja acessado rapidamente, mesmo que existam lacunas significativas entre os valores atribuídos.  
c) JavaScript suporta matrizes esparsas por meio de seu modelo de arrays, que são implementados como objetos, onde os índices não atribuídos não ocupam espaço na memória, sendo tratados como não definidos (undefined) ao serem acessados.(CERTA)  
d) JavaScript trata arrays esparsos como arrays regulares, mas internamente cria uma tabela hash para mapear os índices aos valores, o que melhora o desempenho de acesso a posições vazias.  
e) JavaScript converte arrays esparsos em listas dinâmicas internamente, garantindo que a memória seja alocada apenas para os elementos que contêm valores, enquanto os índices vazios são ignorados.

Quais são as questões de projeto para matrizes?  
a) As principais questões de projeto incluem como as matrizes são alocadas (em tempo de compilação ou execução), se o tamanho pode ser dinâmico ou estático, e como a verificação de limites será realizada, para evitar acessos fora da faixa de índices.(CERTA)  
b) As questões de projeto para matrizes incluem a escolha entre usar ponteiros ou referências para acessar os elementos, além de determinar se as matrizes devem ser implementadas com memória compartilhada para otimizar o uso de recursos.  
c) As questões de projeto incluem determinar se o acesso aos elementos da matriz deve ser feito apenas por índices numéricos ou se suportará também a indexação por strings ou outros tipos de dados dinâmicos e quais as fatias são permitidas.  
d) Uma questão central do projeto de matrizes é definir se todos os elementos devem ter o mesmo tipo de dados ou se podem armazenar múltiplos tipos de dados e quantos níveis de índices são permitidos.  
e) As questões de projeto se concentram na forma como as operações de multiplicação e soma entre matrizes são implementadas diretamente na sintaxe da linguagem, e quais os tipos permitidos para os índices.

Quais são as questões de projeto para os tipos Ponteiro?  
a) A linguagem deve suportar tipo ponteiro, tipo referência ou ambos tipos de dados? Ponteiros devem ser restritos quanto ao tipo de valor para o qual eles apontam, ou seja, tenha um tipo associado que limite o tipo de dados para o qual ele pode apontar?(CERTA)  
b) A linguagem deve permitir que ponteiros sejam usados apenas em funções e não em outras partes do código? Ponteiros devem ser implementados como uma estrutura de dados exclusiva, independente do sistema de tipos?  
c) Os ponteiros devem ser desconsiderados na implementação de algoritmos? A linguagem deve exigir que todos os ponteiros sejam tratados como ponteiros nulos por padrão, para simplificar o processo de alocação de memória e reduzir o risco de vazamentos de memória?  
d) A linguagem deve permitir a conversão automática de ponteiros em inteiros sem restrições, de modo que os desenvolvedores não precisem se preocupar com a segurança dos tipos? A linguagem deve proibir o uso de ponteiros em estruturas de dados complexas, como listas encadeadas ou árvores?  
e) A linguagem deve tratar todos os ponteiros como ponteiros de função? A linguagem deve obrigar os desenvolvedores a declarar todos os ponteiros como variáveis globais, para garantir que sejam acessíveis em qualquer parte do código?

Deixar mais claro o porque a questão e) não é a certa, porque ela não é absurdamente errada, pelo contrário, pode realmente confundir porque tem partes certas. Vou dividi-la em 3 partes.  
  
1) "Utilizar constantes nomeadas ajuda a evitar erros de codificação"  
2) "ao invés de inserir o mesmo valor repetidamente, o programador utiliza um único identificador que representa aquele valor"  
3) "garantindo que ele seja consistente em todo o código."  
  
  
1) Isso não é verdade. Erros de codificação são falhas ou defeitos que ocorrem no código de um programa que podem causar um comportamento inesperado, não desejado ou incorreto no software. Usar constantes nomeadas é somente trocar um valor literal com pouca semântica por um identificador com mais significado visando a melhoria da legibilidade. Se o valor estiver errado por exemplo, não adianta nada nomeá-lo e colocar numa constante que se tiver que acontecer um erro, ele acontecerá com o uso do literal diretamente ou da constante nomeada (como por exemplo atribuir a constante um valor fora do limite do array).   
2) Isso é verdade, mas não é vantagem. É como devemos usar as constantes nomeadas. Como devemos fazer algo não é a vantagem. Vantagem é, caso esse algo seja feito, quais benefícios serão obtidos com isso. Exemplo.: Devemos estudar 1 hora por dia. A vantagem disso será um provável melhor desempenho numa prova (Pegou o código? kkk).  
3) Isso é verdade e é uma vantagem. A constante nomeada ajuda na manutenção do código evitando a geração de inconsistências como, por exemplo, esquecer de mudar um valor literal em todos os lugares que ele é utilizado, melhorando assim a confiabilidade do programa.

Quais são os argumentos contra a representação de valores booleanos como bits únicos em memória?  
a) Usar um único bit, em vez de um byte ou uma palavra inteira, pode ser desvantajoso em sistemas com recursos limitados ou em programas que lidam com grandes quantidades de dados booleanos.  
b) Se há muitos valores booleanos (por exemplo, em um array ou bitmap), representar cada um deles como um bit pode gastar uma quantidade significativa maior de memória.  
c) Em protocolos de comunicação onde a largura de banda é limitada, representar valores booleanos como bits únicos aumenta a quantidade de dados transmitidos, devido ao controle adicional necessário, onerando a comunicação.  
d) Como compiladores não podem utilizar técnicas de otimização que aproveitam a representação de booleanos como bits únicos, agrupar várias variáveis booleanas em um único byte se torna uma tarefa dispendiosa.  
e) Muitas arquiteturas operam em unidades de byte ou múltiplos de bytes. Ao acessar ou modificar um bit único, à necessidade de operações extras para isolar e processar o bit, perdendo assim em simplicidade e desempenho.(CERTA)

Que linguagem suporta matrizes com elementos alocados em índices negativos (e não somente uma convenção para acessar os elementos a partir do final da matriz)?  
a) C#  
b) Ruby  
c) Python  
d) PHP  
e) Fortran 77(CERTA)

Qual a função de acesso para uma matriz bidimensional retangular (m, n)?  
a) localizacao(a[i,j]) = endereco\_de\_a[0,0] + (i \* n + j) \* tamanho\_do\_elemento (CERTA)  
b) localizacao(a[i,j]) = endereco\_de\_a[0,0] + (i + j \* m) \* tamanho\_do\_elemento  
c) localizacao(a[i,j]) = endereco\_de\_a[0,0] + (i \* m + j) \* tamanho\_do\_elemento  
d) localizacao(a[i,j]) = endereco\_de\_a[0,0] + (i \* n + j) \* tamanho\_do\_elemento  
e) localizacao(a[i,j]) = endereco\_de\_a[0,0] + i \* j \* tamanho\_do\_elemento

Defina referências completamente qualificadas para campos em registros.   
a) Referências completamente qualificadas são aquelas que permitem acessar campos de um registro sem precisar especificar o nome do registro, usando apenas o nome do campo, melhorando a legibilidade.  
b) Referências completamente qualificadas são referências que utilizam o caminho completo desde o nome do registro até o campo específico, garantindo que o acesso ao campo seja claro e sem ambiguidades.(CERTA)  
c) Referências completamente qualificadas são usadas em estruturas de dados que não permitem acesso direto aos seus campos.  
d) Referências completamente qualificadas são uma forma de acessar campos de um registro que não requerem a especificação do tipo de dado armazenado no campo.  
e) Referências completamente qualificadas são implementações que otimizam o desempenho do acesso a campos em registros, tornando a execução mais rápida.

Quais vantagens as variáveis de tipo de referência têm em relação aos ponteiros?  
a) As variáveis de tipo de referência garantem que todas as operações de memória sejam realizadas de maneira segura, prevenindo erros de acesso a memória fora dos limites.  
b) Variáveis de tipo de referência podem ser facilmente convertidas em ponteiros, o que melhora a interoperabilidade com linguagens de baixo nível.  
c) O gerenciamento automático de memória através do coletor de lixo reduz o risco de vazamentos de memória em comparação ao gerenciamento manual de ponteiros.(CERTA)  
d) Variáveis de tipo de referência não podem ser nulas, o que elimina o risco de erros de ponteiro nulo em tempo de execução.  
e) Variáveis de tipo de referência são sempre mais eficientes em termos de desempenho do que ponteiros, pois não exigem operações adicionais de desreferenciação.

O que foi perdido na decisão dos projetistas de Java de não incluírem os ponteiros?  
a) A flexibilidade de controlar a passagem de parâmetros por valor ou por referência, reduzindo a sobrecarga no desempenho ao evitar cópias desnecessárias de objetos.  
b) O suporte à aritmética de ponteiros, permitindo operações matemáticas diretamente nos endereços de memória para acessar e modificar dados de maneira mais eficiente.  
c) A capacidade de manipular diretamente a memória através de endereços, permitindo operações de baixo nível e controle preciso sobre a alocação e desalocação de memória. (CERTA)  
d) A capacidade de implementar estruturas de dados mais avançadas, como listas encadeadas e árvores, sem a necessidade de bibliotecas adicionais.  
e) O controle direto sobre o gerenciamento de memória, eliminando a dependência do coletor de lixo e permitindo a liberação explícita de memória em tempo real.

O que foi perdido na decisão dos projetistas de Java de não incluírem os ponteiros?  
a) A flexibilidade de controlar a passagem de parâmetros por valor ou por referência, reduzindo a sobrecarga no desempenho ao evitar cópias desnecessárias de objetos.  
b) O suporte à aritmética de ponteiros, permitindo operações matemáticas diretamente nos endereços de memória para acessar e modificar dados de maneira mais eficiente.  
c) A capacidade de manipular diretamente a memória através de endereços, permitindo operações de baixo nível e controle preciso sobre a alocação e desalocação de memória.  
d) A capacidade de implementar estruturas de dados mais avançadas, como listas encadeadas e árvores, sem a necessidade de bibliotecas adicionais.  
e) O controle direto sobre o gerenciamento de memória, eliminando a dependência do coletor de lixo e permitindo a liberação explícita de memória em tempo real.

O que são parâmetros formais de subprogramas?  
a) Parâmetros formais são variáveis locais que armazenam temporariamente os resultados de um subprograma após sua execução.  
b) Parâmetros formais são valores definidos pelo subprograma ao final de sua execução, que serão retornados ao código chamador.  
c) Parâmetros formais são as variáveis declaradas no subprograma que representam os valores que serão passados pelos argumentos durante a chamada do subprograma. (CERTA)  
d) Parâmetros formais são variáveis globais usadas por subprogramas para compartilhar dados entre diferentes partes do programa.  
e) Parâmetros formais são constantes definidas dentro do subprograma que servem para controlar seu fluxo de execução.

Quais são as vantagens e desvantagens de parâmetros de subprogramas com palavras-chave?  
a) A vantagem dos parâmetros com palavras-chave é que eles permitem reordenar os argumentos na chamada do subprograma, o que pode facilitar a legibilidade do código. A desvantagem é que a sintaxe se torna mais complexa, exigindo que todos os parâmetros sejam passados na ordem correta.  
b) Uma vantagem dos parâmetros com palavras-chave é que eles permitem a passagem de apenas alguns argumentos, omitindo os outros. A desvantagem é que isso impede o uso de valores padrão nos parâmetros.  
c) A principal vantagem dos parâmetros com palavras-chave é que eles garantem que os argumentos sejam passados por referência, melhorando a eficiência. A desvantagem é que não permitem o uso de parâmetros variáveis.  
d) A vantagem dos parâmetros com palavras-chave é que eles tornam o código mais legível ao identificar claramente qual valor é passado para qual parâmetro, e a desvantagem é que podem tornar a chamada do subprograma mais longa e verbosa. (CERTA)  
e) Os parâmetros com palavras-chave têm a vantagem de melhorar o desempenho da execução do subprograma, mas a desvantagem é que eles não podem ser usados em subprogramas que aceitam um número variável de parâmetros.

Que linguagens permitem um número variável de parâmetros de entrada em subprogramas?  
a) Python, C# e Ruby. (CERTA)  
b) Pascal, Java e PHP.  
c) Cobol, Rust e JavaScript.  
d) Ada, C++ e Swift.  
e) Fortran, Kotlin e Perl.

Qual é a principal diferença entre uma função e um procedimento em linguagens de programação?  
a) Uma função pode ser chamada de qualquer lugar no código, enquanto um procedimento só pode ser chamado dentro de outras funções.  
b) Uma função sempre retorna um valor, enquanto um procedimento não retorna nenhum valor. (CERTA)  
c) Uma função é utilizada exclusivamente para operações matemáticas, enquanto um procedimento é utilizado para manipulações de dados.  
d) Um procedimento não pode ter parâmetros, enquanto uma função deve ter pelo menos um parâmetro.  
e) Um procedimento pode retornar múltiplos valores, enquanto uma função só pode retornar um valor.

Que linguagens permitem que as definições de subprogramas sejam aninhadas?  
a) Python e C#. (CERTA)  
b) Kotlin e PHP.  
c) JavaScript e Objective-C.  
d) Java e Swift.  
e) Rust e C++.

Qual das alternativas representa duas características gerais dos subprogramas (exceto co-rotinas)?   
a) Subprogramas herdam o contexto de execução de onde são chamados, sem a necessidade de isolamento de variáveis e s são definidos para executar uma tarefa específica.  
b) Subprogramas podem ser chamados com diferentes tipos de parâmetros sem a necessidade de sobrecarga e eles devem possuir um número fixo de parâmetros.  
c) Subprogramas não podem modificar os valores das variáveis globais de um programa já sempre trabalham com cópias dos dados e eles devem ser declarados antes da sua primeira utilização no código.  
d) Cada subprograma tem um único ponto de entrada (o início do código do subprograma) e seu controle sempre retorna ao chamador quando a execução do subprograma termina. (CERTA)  
e) Subprogramas devem ser definidos antes de qualquer execução uma vez que eles não podem ser definidos de maneira dinâmica ou durante o tempo de execução e eles podem ser reutilizados várias vezes.

Quais são os três modelos semânticos de passagem de parâmetros para subprogramas?   
a) Passagem por modo saída, modo entrada, e modo entrada/saída.  
b) Passagem por resultado, por valor, e por arranjo.  
c) Passagem por valor, por referência e por nome.  
d) Passagem direta, indireta e por retorno  
e) passagem por valor, por condição e por contexto.

Quais a seguir são modelos de implementação de passagem de parâmetros para subprogramas?   
a) Passagem por modo saída, modo entrada, e modo entrada/saída.  
b) Passagem por resultado, por valor, e por arranjo.  
c) Passagem por valor, por referência e por nome.  
d) Passagem direta, indireta e por retorno  
e) passagem por valor, por condição e por contexto.

O que é um subprograma sobrecarregado?  
a) Um subprograma sobrecarregado é aquele que pode ser interrompido e retomado em um ponto posterior de sua execução.  
b) Um subprograma sobrecarregado é um subprograma que pode ser chamado simultaneamente por vários threads sem causar conflitos de acesso aos seus dados.  
c) Um subprograma sobrecarregado é um subprograma que pode ser definido com diferentes números ou tipos de parâmetros, mas mantém o mesmo nome.  
d) Um subprograma sobrecarregado é aquele que contém várias instruções de loop aninhadas e sobrecarrega o processador com cálculos complexos.  
e) Um subprograma sobrecarregado é um subprograma que herda propriedades e métodos de outro subprograma com o mesmo nome em uma hierarquia de classes.

O que é polimorfismo paramétrico?  
a) Polimorfismo paramétrico ocorre quando um subprograma pode tomar diferentes números de parâmetros em chamadas sucessivas, variando de acordo com o contexto de execução.  
b) Polimorfismo paramétrico é uma técnica que permite que subprogramas possam operar sobre parâmetros de diferentes tipos, sendo os tipos definidos de forma genérica no momento da declaração.  
c) Polimorfismo paramétrico permite que um subprograma seja definido em tempo de execução com parâmetros que mudam conforme a execução do programa.  
d) Polimorfismo paramétrico ocorre quando um subprograma pode retornar diferentes tipos de dados dependendo do valor de entrada, com base em verificações dinâmicas.  
e) Polimorfismo paramétrico permite que o compilador decida dinamicamente qual tipo de parâmetro usar em um subprograma, sem que seja necessário declarar o tipo explicitamente em tempo de compilação.

Que linguagens permitem que o usuário sobrecarregue operadores?  
a) Visual Basic e Swift.  
b) Pascal e Rust.  
c) Python e JavaScript.  
d) C++ e Kotlin.  
e) Java e C Sharp.

De que maneiras as corrotinas são diferentes dos subprogramas convencionais?  
a) Corrotinas não mantêm o estado entre as chamadas, enquanto subprogramas convencionais preservam o contexto de execução após cada chamada.  
b) Corrotinas são subprogramas especiais que tem múltiplas entradas e podem ser usadas para oferecer execução intercalada de subprogramas.  
c) Corrotinas têm um único ponto de entrada e vários pontos de saída, enquanto subprogramas convencionais têm múltiplos pontos de entrada e um único ponto de saída.  
d) Corrotinas exigem mais recursos do sistema em comparação com subprogramas convencionais, que são mais leves em termos de uso de memória.  
e) Corrotinas precisam ser definidas antes da execução do programa, enquanto subprogramas convencionais podem ser definidos durante a execução.

Descreva o problema de passar matrizes multidimensionais como parâmetros.   
a) O maior problema consiste na elaboração da função de associação de armazenamento, usada para associar valores de índice de elementos da matriz a endereços na memória, de forma que seja flexível, reutilizável, legível e que não seja propensa a erros.  
b) Todas as linguagens de programação permitem a passagem de matrizes multidimensionais por referência, sem a necessidade de especificar o tamanho da matriz, neste caso o acesso a elementos de matrizes multidimensionais é sempre garantido por verificações automáticas de limites, evitando erros de acesso fora dos limites.  
c) A passagem de matrizes multidimensionais requer que o programador especifique as dimensões da matriz, o que pode levar a erros de incompatibilidade entre as dimensões passadas e as esperadas pelo subprograma, especialmente em linguagens que não suportam arrays dinâmicos.  
d) Quando matrizes multidimensionais são passadas como parâmetros, é comum que o subprograma que as recebe tenha que alocar memória adicional para armazenar a matriz, o que pode levar a vazamentos de memória se não for tratado adequadamente.  
e) A passagem de matrizes multidimensionais geralmente resulta em cópias completas da matriz, aumentando o uso de memória e o tempo de processamento, pois não há uma maneira padronizada de passar referências para o conteúdo da matriz.

Quais são as questões de projeto referentes a subprogramas?  
a) Quais variáveis globais devem ser utilizadas dentro do subprograma? É importante determinar se as variáveis globais devem ser alteradas ou apenas lidas, sem considerar a passagem de parâmetros? A escolha do tipo de retorno do subprograma deve ser feita sem considerar os tipos de parâmetros?   
b) O número de subprogramas que devem ser criados deve ser baseado na complexidade do código principal, sem qualquer consideração sobre a passagem de parâmetros ou tipos de dados? A prioridade deve ser apenas a eficiência do código, sem se preocupar com a legibilidade? Qual a melhor maneira de implementar subprogramas como métodos de extensão?  
c) Qual método ou métodos de passagem de parâmetros a ser utilizado? Os tipos reais devem ser verificados em relação aos tipos de parâmetros formais? Caso o subprograma pode ser passado como parâmetro, qual o ambiente de referência dele?  
d) O foco deve ser apenas na adição de funcionalidades, sem levar em conta os parâmetros e os tipos de dados envolvidos? Como determinar a profundidade máxima de recursão que o subprograma pode ter? A avaliação deve ser baseada no tamanho do código e na eficiência, sem levar em conta a passagem de parâmetros?   
e) A escolha do nome do subprograma deve ser feita exclusivamente com base na convenção de nomenclatura da linguagem utilizada, sem considerar a funcionalidade ou os parâmetros que ele receberá? Qual deve ser a documentação do subprograma, focando apenas nas explicações sobre a lógica interna, sem se preocupar com os tipos de parâmetros e suas implicações na chamada? Como o subprograma deve gerenciar o escopo das variáveis locais, priorizando a utilização de variáveis globais em vez de discutir a passagem de parâmetros e suas verificações?

Quais são as duas questões que surgem quando nomes de subprogramas são parâmetros?  
a) Verificação do ambiente de execução correto para o subprograma e a checagem de compatibilidade entre os tipos das variáveis locais e globais usadas no subprograma.  
b) Verificação de tipo dos parâmetros das ativações do subprograma passado como parâmetro e ambiente de referenciamento correto (escopo) para executar o subprograma passado.  
c) Checagem de tipos entre o subprograma original e o subprograma chamado e o ambiente de armazenamento necessário para o subprograma ser ativado corretamente.  
d) Definição de quais variáveis globais podem ser referenciadas pelo subprograma e verificação de tipos entre o subprograma chamado e as variáveis locais no escopo atual.  
e) Verificação do ambiente de execução correto para as variáveis passadas como parâmetro e definição do contexto em que o subprograma deve retornar após sua execução.