1 Introduction, History

1. true / false Invisible things are not important.
2. true / false Computer science is the study of computers.
3. true / false FORTRAN can reasonably be considered the first programming language.
4. true / false Fortran, before FORTRAN 90, had no recursion and no pointers.
5. true / false Frege contributed to the mathematical foundations of the theory of quantification.
6. true / false Cambridge prefix notation cannot be used to expression assignment or loops.
7. true / false J is to APL as Beta is to SIMULA.
8. true / false HTML is a programming language.
9. true / false Interactive language systems execute programs slower than compiled programs.
10. true / false ALGOL can reasonably be considered the most influential programming language.
2 Syntax

1. true / false  Lexical analysis determines the phrase structure of a language’s tokens.

2. true / false  Regular expressions are great because they are more expressive than other common ways to define formal languages.

3. true / false  The Cherokee script is used in writing FORTRAN programs.

4. true / false  A formal language is a set of symbols from an alphabet.

5. true / false  Formal language theory applies to the lexical structure of programming languages, but not to the phrase structure.

6. true / false  It is impossible to automatically generate a program to do lexical analysis from a description of the tokens.

7. true / false  An ambiguous grammar and an unambiguous grammar may both describe the same language.

8. true / false  BNF describes more formal languages than does regular expressions.

9. true / false  If a greedy quantifier matches some string, then the corresponding possessive quantifier will match too, although possibly matching fewer symbols.

10. true / false  There are tools in wide-spread use to generate parsers automatically from their descriptions.

11. true / false  There are tools in wide-spread use to generate scanners automatically from regular expressions.
3 Semantics

1. true / false A loop invariant is a value that does not change.

2. true / false An assertion is the same as a condition.

3. true / false If $P \Rightarrow Q$, then the set of computer states characterized by $Q$ is a subset of the set of state characterized by $P$.

4. true / false “Sue me if my precondition is false, sue you if my postcondition is false.”

5. true / false A significant weakness of denotational semantics is with concurrent programming.

6. true / false “Sue me if my postcondition is false, sue you if my precondition is false.”

7. true / false The propositional formula $A \& B \Rightarrow C$ is necessarily true if $B$ is false.
4 Types

1. true / false All languages take basically the same approach to types.
2. true / false A program that cannot be statically type has a type insecurity.
3. true / false Bounded quantification polymorphism is a combination of ad hoc and universal polymorphism.
4. true / false C uses name equivalence.
5. true / false Implicit coercion is an example of universal polymorphism.
6. true / false It is possible to statically type-check heterogeneous, composite data types with dynamic access.
7. true / false Java uses structural equivalences of types.
8. true / false Modern languages tend to favor structural equivalence of types.
9. true / false Modula-3 uses name equivalence.
10. true / false Java uses name equivalence.
11. true / false Parametric universal polymorphism is the basis of object-oriented programming.
12. true / false Structural equivalence is necessary to strengthen type abstraction.
13. true / false An Ada subtype is a type.
14. true / false An Ada subtype gives the programmer a compile-time guarantee about the behavior of the program.
15. true / false Polymorphism means “many forms.”
16. true / false Variant records cannot be statically typed.
17. true / false Ada has inheritance as in OO languages using tagged record types.
18. true / false “Branding” as in Modula-3 allows the programmer to make structural equivalence when the language supports name equivalence.
19. true / false A characteristic of universal polymorphism is an infinite number of possibilities.
20. true / false The type operator for arrays in Java is covariant.
21. true / false Arrow (function) types are, by their nature, contravariant in the domain and covariant in the range.

22. true / false A type insecurity arises when the data is misinterpreted.

23. true / false A type variable stands in place of a specific type.

24. true / false Ada uses name equivalence.

25. true / false Any binary file can be interpreted as a US-ASCII text file.

26. true / false The array type operator in C# is covariant.

27. true / false The same sequence of bits can mean different things.

28. true / false Universal, parametric polymorphism takes advantage of the fact that many programs don’t care about the specific types of its data.
5 Blocks

1. true / false  Fortran has two different kinds of subprogram declarations: function and subroutine.

2. true / false  Ada uses call-by-name parameter passing.

3. true / false  Localization of scope means assigning to variable exactly once.

4. true / false  There is only one environment in which a procedure is called.

5. true / false  ALGOL uses call-by-name parameter passing.

6. true / false  All local variables are local to some block.

7. true / false  Static scoping is the same as dynamic scoping for local variables.

8. true / false  Non-local variable access in a block-structured language can be implemented in a single machine instruction.

9. true / false  The stack is generally considered to be a very efficient way of storing local variables.

10. true / false  With just two values a non-local variable reference can be found.

11. true / false  There is only one environment in which a procedure is defined.
6 Logic Programming

1. true / false All formulas in first-order predicate logic can be represented by Prolog clauses.

2. true / false The unification problem has $O(n)$ time complexity.

3. true / false There exists a $O(n)$ algorithm for unification.

4. true / false Like Haskell, Prolog uses arrays for complex data structures.

5. true / false A literal can be both a fact and a query in Prolog.

6. true / false There are no implementations of Prolog.

7. true / false Conjunction is denoted by a comma in Prolog.

8. true / false Functors are distinguished syntactically from predicates in Prolog.

9. true / false Functors are used to define functions in Prolog.

10. true / false Functors have to be declared before they are used in Prolog.

11. true / false Omitting the occurs-check makes Prolog logically sound.

12. true / false The order of the rules in a Prolog program is a factor in determining the number of solutions in the search space.

13. true / false Prolog builds a search space from the query, and then searches it.

14. true / false Prolog does a breadth-first search of the search space.

15. true / false Prolog has construct specifically for functions.

16. true / false ISO standard Prolog uses the occurs check.

17. true / false Unification requires $O(n^2)$ running time.

18. true / false The only practical algorithm for unification is $O(n^2)$.

19. true / false A Prolog implementation requires garbage collection.

20. true / false A Prolog query may have an infinite number of solutions, yet Prolog may find none of them.

21. true / false A Prolog search space may have an infinite number of solutions.

22. true / false A different query gives rise to a different Prolog search space.

23. true / false Prolog atoms are nullary functor symbols.
24. true / false Prolog backtracks when no way can be found to make progress on the first goal.

25. true / false It is impossible to compile Prolog.

26. true / false Prolog has relations.

27. true / false Prolog is Turing complete.

28. true / false Prolog relations can be defined recursively.

29. true / false In Prolog, a solution is found when all the goals have been established.

30. true / false Logic investigates and classifies the structure of statements and arguments.

31. true / false Prolog uses depth-first search (DFS) when searching for a solution.

32. true / false Functors in Prolog require heap allocation.

33. true / false Recursion in Prolog does require allocation of memory in the heap.

34. true / false The order in which Prolog clauses are asserted matters a great deal.
7 Functional Programming

1. true / false In Haskell backquotes make an identifier of a binary function parsed as infix.

2. true / false In Haskell every binary function name can be used in infix form (function in the middle).

3. true / false In Haskell every binary function name can be used in prefix form (function out front).

4. true / false Function arguments are separated by commas in Haskell.

5. true / false Functional programming is characterized by Cambridge prefix notation.

6. true / false Functional programming is slow because it is usually interpreted.

7. true / false Haskell has an assignment, if, and while statement (but these are rarely used).

8. true / false In this class we will study the lambda calculus—the theory behind functional programming languages.

9. true / false A Haskell implementation will use a stack of activation records at runtime for non-local variable access like Algol-like languages.

10. true / false The implementation of a functional language, like Haskell or Scheme, will use a heap at runtime like Algol-like languages.

11. true / false A canonical value is one which can be rewritten or simplified.

12. true / false A subprocedure that takes another subprocedure as an argument is said to be higher-order.

13. true / false All functions have names in Haskell.

14. true / false All functions in Haskell are higher-order.

15. true / false In an eager language one can define infinite data structures directly.

16. true / false In Haskell the index function (!!) is O(1).

17. true / false It is not possible to read and write using standard IO in Haskell.

18. true / false Haskell is an example of a non-procedural language.

19. true / false Parentheses are part of the syntax of function calls in Haskell.
20. true / false  Destructors are commonly used in Haskell.
21. true / false  Function application is left associative.
22. true / false  The programmer is required to declare the types of functions in Haskell.
23. true / false  Haskell is always interpreted.
24. true / false  Lists are immutable data structures in Haskell.
25. true / false  Lists of functions are possible in Haskell.
26. true / false  The function space type operator is right associative.
27. true / false  Type names are in lower-case in Haskell.
28. true / false  Visual Studio supports Haskell.
29. true / false  All functions are canonical values. [hnf]
30. true / false  Constructors can be used in Haskell function definitions as patterns.
31. true / false  Functions can take tuples as an argument in Haskell.
32. true / false  Functions have one argument in Haskell.
33. true / false  Haskell is Turing complete.
34. true / false  Haskell is a language (one of the few) whose name is taken from the given or first name of a real person.
35. true / false  Haskell is lazy.
36. true / false  A canonical value is one which can be rewritten or simplified.
37. true / false  All functions have names in Haskell.
38. true / false  A subprocedure that takes another subprocedure as an argument is said to be higher-order.
39. true / false  All functions in Haskell are higher-order.
40. true / false  In Haskell \( \lambda x \rightarrow x(2) \) is a higher-order function.
41. true / false  In Haskell \( \lambda x \rightarrow x(2) \) is an anonymous function.
42. true / false  In a lazy language one can define infinite data structures directly.
43. true / false  Lists of functions are possible in Haskell.
44. true / false  Scheme is LISP cleaned up.
45. true / false The Glasgow Haskell Compiler has an interactive interface.

46. true / false The abstract data type for lists is predefined in Haskell.

47. true / false The types of high-order functions must have more than one \( \rightarrow \) in them.

48. true / false The most important data structure in functional languages is the immutable list.

49. true / false Like most imperative languages, Scheme and OCaml evaluates function arguments before passing them to a function.

50. true / false A (function without side effects) is said to be strict if it is undefined (fails to terminate) when any of its arguments is undefined.

51. true / false Tuples of functions are possible in Haskell.

52. true / false Type class constraints in Haskell appear before the \( \rightarrow \).

53. true / false Type variables begin with a lower-case letter in Haskell.