High Level Programming Language 1 QUES. Bank

s m sadman sakib sayor,

bsc cs, university of debrecen.

Chapter 1.Past papers

Chapter 2.Preliminaries

Chapter 3.Describing syntax and semantics

Chapter 4.Lexical and syntax analysis

Chapter 5.Names,binding and scope

Chapter 6.Data types

Chapter 7.Expression and assignment statements

Chapter 8.Statement level control structure

Chapter 9.Sub-program

Chapter 10.Multiple Choice Questions

**PREVIOUS PAPER**

**Chapter 1 – Past papers**

1.Define lexical and syntax analysis with a few sentences(2 points)

Lexical analysis: It is essentially a pattern matcher and serves as the front end of a syntax analyzer. It collects lexemes (collects characters into logical groupings) from a given input string and produce the corresponding tokens (internal codes’ categories).

Syntax analysis: the process of analyzing syntax. It serves two purposes: 1. Make sure the syntax of the input program is syntactically correct. 2. If the grammar input is correct, it will generate a complete parse tree.

2. What are the primary tasks of a lexical analyzer? (3 points)

(Describe it with your own words, what steps are performed and what produced by it)

Primary tasks: It helps to convert a sequence of characters into a sequence of tokens.

Steps: 1. It collects characters into logical groupings and assigns internal codes to the groupings according to their structure.

2. It extract lexemes from a given input string and produce the corresponding tokens.

3. It skipping comments and white space outside lexemes.

3.Which is faster in execution: a compiler or a pure interpreter?(4 points)

Explain your answer.

* **Compiler implementation,** in this method programs can be translated into machine language, which can be executed directly on the computer. This method has the advantage of **very fast** program execution, once the translation process is complete.
* **Pure interpretation**. It lies at the opposite end (from compilation) among implementation methods. With this approach, programs are interpreted by another program called an interpreter, with no translation whatever. The interpreter program acts as a software simulation of a machine whose fetch-execute cycle deals with high- level language program statements rather than machine instructions. This software simulation obviously provides a virtual machine for the language.

(Compiler is faster) The pure interpreter is 10-100 times slower than in compiled system, because of the decoding of the high-level language statements, which are far more complex than machine language instructions.

4.Define the meaning of syntax and semantics.(2 Points)

-Syntax: The form or structure of the expressions, statements and program units.

-Semantics: The meaning of the expressions, statements, and program units.

\*Syntax and semantics provide a language’s definition.

5.How you can describe a variable? What is variable?

[list and explain all the attributes of a variable (lifetime and scope and excl)

Variable is an abstraction of a computer memory cell or collection of cells, Programmers often think of variables as names For memory locations, but there is much more to a variable than Just a name. A variable can be characterized as a sextuple of attributes: (name, address, value, type, lifetime, and scope).

Name: Variable names are the most common names in programs. Not all variables have them Address: The address of a variable is the machine memory address with which it is associated.

Type: The type of a variable determines the range of values the variable can store and the set of operations that are defined for values of the type.

Value: The value of a variable is the contents of the memory cell or cells associated with the variable.

Lifetime: The lifetime of a variable is the time during which the variable is bound to a specific memory location.

Scope: The scope of a variable is the range of statements in which the variable is visible.

6.What is the definition of block?

A method of creating static scopes inside program units is called blocks. A block is a section of code that has its own local variable whose scope is minimized, A block is specified in the C-based languages as a compound statement that begins with one or more data definitions. In C programming language block is created by using a pair of Curly bract. The beginning of a block is denoted by an open curly brace ‘{‘and the end is denoted by closing curly bract’}’

7.What does it mean when a variable is static?

[Explain the advantage and disadvantage]

Static variables are those that are bound to memory cells before program execution begins and remain bound to those same memory cells until program execution terminates.

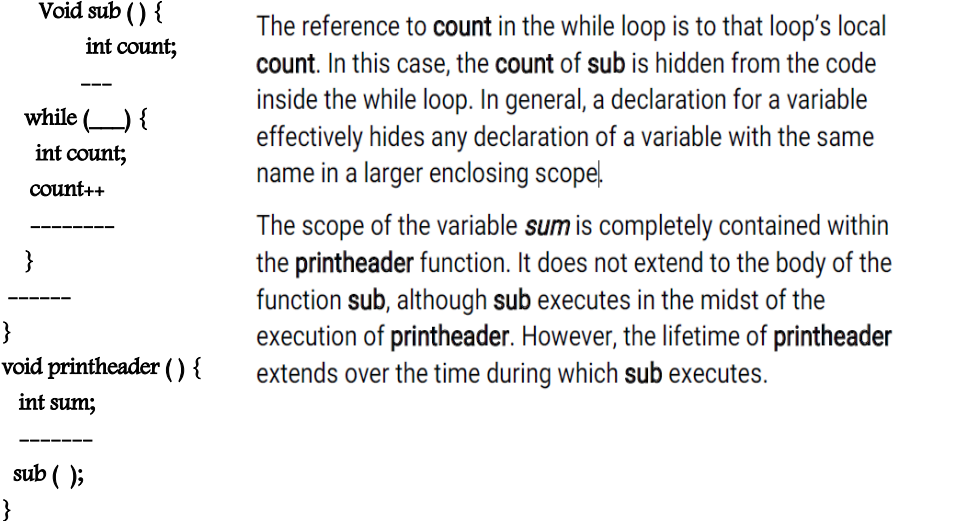
Advantages: (efficiency) All addressing of static variables can be direct; other kinds of variables often require indirect addressing, which is slower. Also, no run-time overhead is incurred for allocation and deallocation of static variables, although this time is often negligible.

Disadvantage: reduced flexibility; in particular, a language that has only static variables cannot support recursive subprograms. Another disadvantage is that storage cannot be shared among variables.

8.Define scope and lifetime! Explain C’s scoping and lifetime in the following code fragment:

- Scope: The range of statements in which the variable is visible. A variable is visible in a statement if it can be referenced in that statement.

-Lifetime: A time during which the variable is bound to a Specific memory location. The life-time begins when it is Bound to a specific cell and ends when it is unbound from That cell.

.

9.What are the design issues for multiple-selection statements?

[Explain you answer for C’s array]

1-What is the form and type of the expression that controls the selection?

2- How are the selectable segments specified?

3- Is execution flow through the structure restricted to include just a single selectable segment?

4- How are the case values specified?

5- How should unrepresented selector expression values be handled, if at all?

The C switch statement has virtually no restrictions on the placement of the case expressions, which are treated as if they were normal statement labels. This laxness can result in highly complex structure within the switch body.

10.How does C support relation and Boolean expressions?

Relation: The syntax of the relational operators for equality and inequality differs among some programming languages. For example, for inequality, the C-based languages use !=

Boolean: The precedence of the arithmetic, relational, and Boolean operators in the C-based languages is as follows: Highest postfix ++, -- unary +, unary -, prefix ++, --, ! \*, /, % binary +, binary - , <=, >= =, != && Lowest ||

11.What are the two common problems with pointers? [Explain your answer]

Dangling Pointers: First, the location being pointed to, may have been reallocated to some new heapdynamic variable. If the new variable is not the same type as the old one, type checks of uses of the dangling pointer are invalid. Even if the new dynamic variable is the same type, its new value will have no relationship to the old pointer’s dereferenced value.

Lost Heap-Dynamic Variables: A lost heap-dynamic variable is an allocated heap-dynamic variable that is no longer accessible to the user program. Such variables are often called garbage, because they are not useful for their original purpose, and they also cannot be reallocated for some new use in the program.

12.Define the subprogram protocol?

The protocol of a subprogram is its parameter profile plus, if it is a function, its return type. In languages in which subprograms have types, those types are defined by the subprogram’s protocol.

13.Define the referencing environment?

The referencing environment of a statement is the collection of all variables that are visible in the statement. The referencing environment of a statement in a static-scoped language is the variables declared in its local scope + the collection of all variables of its ancestor scopes that are visible.

14.What does it mean when a variable is implicit heap dynamic? [Explain its advantages and disadvantage]

Implicit heap-dynamic variables are bound to heap storage only when they are assigned values. all their attributes are bound every time they are assigned.

Advantage: they have the highest degree of flexibility, allowing highly generic code to be written. Disadvantage:

1-the run-time overhead of maintaining all the dynamic attributes,

2- the loss of some error detection by the compiler.

Explicit heap dynamic: allocated and deallocated by explicit directives which is specified by the programmer. Advantage: providing dynamic storage management. Disadvantage: inefficient and unreliable

15.What are the design issues for arrays? [With C]

1-What types are legal for subscripts?

2-Are subscripting expressions in element references range checked?

3-When are subscript ranges bound?

4-When does array allocation take place?

5-Are ragged or rectangular multi dimensioned arrays allowed, or both?

6-Can arrays be initialized when they have their storage allocated?

7-What kinds of slices are allowed, if any?

16.What is unusual about C’s multiple selection statement?

The C switch statement has virtually no restrictions on the placement of the case expressions, which are treated as if they were normal statement labels. This laxness can result in highly complex structure within the switch body.

17.What are the three semantic models of parameter passing?

[Explain the pass-byvalue-result model in details]

1-in -mode: Formal parameter can receive data from actual parameter.

2-out-mode: Formal parameter can transmit data to Actual parameter.

3-inout-mode: Do both

Pass-by-value- result is an implementation model for in out -mode Parameters in which actual values are copied. It is in effect a combination of pass- by-value and pass-by-result. The value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable. Pass-by-value-result is sometimes called pass-bycopy, because the actual parameter is copied to the formal parameter at subprogram entry and then copied back at subprogram termination.

the disadvantages of requiring multiple storage for parameters and time for copying values. It shares with pass-by-result the problems associated with the order in which actual parameters are assigned.

the advantage Duplicate space is not required, and no copying is required.

18.Define the subprogram signature?

The parameter profile of a subprogram contains the number, order, and types of its formal parameters.

19.How does short-circuit expression evaluation works?

A short-circuit evaluation of an expression is one in which the result is determined without evaluating all of the operands and/or operators. For example, the value of the arithmetic expression (13 \* a) \* (b / 13 - 1) is independent of the value of (b / 13 - 1) if a is 0, because 0 \* x = 0 for any x. So, when a is 0, there is no need to evaluate (b / 13 - 1) or perform the second multiplication.

20.How is lexical analyzer is part of the syntax analyzer ?

A lexical analyzer serves as the front end of a syntax analyzer. Technically, lexical analysis is a part of syntax analysis. A lexical analyzer performs syntax analysis at the lowest level of program structure. The syntax analyzer takes the lexical units from the lexical analyzer and uses them to construct hierarchical structures called parse trees.

21.Define actual parameter and formal parameter:

Actual Parameters: The values/variables passed while calling a function are called actual parameters.

Formal Parameters: These are the variables written/declared in function definition/prototype, and receive their values when a call to that function is made.

22.What is a stack dynamic variable?

are those whose storage bindings are created when their declaration statements are elaborated, but whose types are statically bound.

23.The difference between lexical and syntax analysis?

1. Simplicity: Techniques for lexical analysis are less complex than those required for syntax analysis, so the lexical-analysis process can be simpler if it is separate.

2. Efficiency: Although it pays to optimize the lexical analyzer, because lexical analysis requires a significant portion of total compilation time.

3. Portability: Because the lexical analyzer reads input program files and often includes buffering of that input, it is somewhat platform dependent. However, the syntax analyzer can be platform independent.

24.what is the generic subprogram and overloaded subprogram?

-Generic is one whose computation can be done on data of different types in different calls.

-overloaded subprogram is one that has the same name as another subprogram in the same referencing environment

25.what is the coroutine?

A coroutine is a special kind of subprogram. Rather than the master-slave relationship between a caller and a called subprogram that exists with conventional subprograms, caller and called coroutines are more equitable. often called the symmetric unit control model.

26. what is operator overload?

It’s a potential problem in which a single operator symbol has more than one meaning.

27.What is reserved word?

is a special word of a programming language that cannot be used as a name. There is one potential problem with reserved words: If the language includes a large number of reserved words, the user may have difficulty making up names that are not reserved.

28.What is closure subprogram?

is a nested subprogram and its referencing environment, which together allow the subprogram to be called from anywhere in a program.

29. Explain how the coercion rules of a language affects its error detection.

Coercion is an automatic conversion of variables. For example, if we want to add int variable to a float variable, the value of int variable will be coerced to float to add them.

So, coercion can result in a loss of error detection.

For example, if we wanted to add a variable x to variable y, that are both int, but if we mistyped and wrote variable w(which is a float) instead of y. The program would not detect the error, and the value of variable x will be just coerced to float.

30.Pass-by-value and pass-by-reference of the code

When the variable is **passed-by-value**, the copy of the actual parameter’s value is stored in the memory, and the called function and caller have two independent variables with the same value. But the caller cannot see modification by the called function.

When the variable is **passed-by-reference,** the memory address of that variable is passed directly to the function. Called function and the caller run the same variable, so if the called function makes any modifications, it is also seen by the caller, that is, it affects its variable.

So, if we pass-by-value the output will be:

50, 60

Because when we call the procedure f, it creates copies of values i and j and changes them without modifying actual values.

So, x will be 10 and y will be 160 inside the procedure f.

But in the main() i and j are not changed and remain the same.

And if we pass-by-reference, instead of creating copies of i and j, procedure f uses their actual addresses, so it modifies the actual values.

So, after we call f (i,j), the output will be:

10, 160

31.Which variable has the longest scope?

*int a;*

*int main()*

*{*

*…*

*int b;*

*}*

*int c;*

The variable 'b' is limited to the main() function and 'c' is accessible only after its declaration. Hence, the variable 'a' has the longest scope when compared with other variables.

32.Dynamic type binding is closely related to implicit heap-dynamic variables. Explain this relationship.

First version: Implicit heap-dynamic variables acquire types only when assigned value, which must be at runtime. Therefore, these variables are always dynamically bound to types.

Choose one version

Second version: Variable a gets its type at runtime, i.e. dynamic type binding, when a value is assigned to it. Since variable a may often change the types of values it holds, hence the memory locations where the values are stored, a must be a reference variable and its dynamically assigned value must be in a dynamically allocated memory location, i.e. the heap.

33.What is the l-value of variable, what is the r-value.

**Main answer:**

The address of a variable is sometimes called its **l-value** because the address is what is required when the name of a variable appears in the left side of an assignment.

A variable’s value is sometimes called its **r-value** because it is what is required when the name of the variable appears in the right side of an assignment statement.

**Just more info:**

A variable can be characterized as a sextuple of attributes: (name, address, value, type, lifetime, and scope).

The address of a variable is the machine memory address with which it is associated.

The value of a variable is the contents of the memory cell or cells associated with the variable.

To access the r-value, the l-value must be determined first.

34.Write a simple assignment statement with one arithmetic operator in some language you know.

For each component of the statement list the various bindings that are required to determine the semantics when the statement is executed.

For each binding, indicate the binding time used for the language.

The **symbol "="** is called as the assignment operator

**My answer:**

(C++)

x = a + b;

Identifiers: x, a , b

Operators: +, =

The type of x, a, b is bound at compile time.

The set of possible values of a, b, x is bound at compiler design time.

For the meaning of + operator: compiler time binding (when types of operands have been determined)

For the value of x, a, b: runtime binding

= is the assignment operator.

Statement is terminated with semicolon ;

**Some answer from book:**

C++ assignment statement:

count = count + 5;

Some of the bindings and their binding times for the parts of this assignment statement are as follows:

• The type of count is bound at compile time.

• The set of possible values of count is bound at compiler design time.

• The meaning of the operator symbol + is bound at compile time, when the types of its operands have been determined.

• The internal representation of the literal 5 is bound at compiler design time.

• The value of count is bound at execution time with this statement.

**Some answer from internet:**

Given the following simple assignment statement in Java

a = b + c

The statement contains the following components:

Identifies: a, b, c

Operators: =, +

For a, b .c:

Type: compiler time binding

Address: if static, loading time binding, otherwise runtime binding

Value: runtime binding

Scope: compiler time binding since Java uses static scoping

Life time: if static, same as the program; if instant variables, same as the object; if local variables of a method, as long as a method call.

For the meaning of the = assignment operator:

Language design time binding

For the meaning of the + operator:

Compiler time binding to sum or concatenation.

35.What is an overloaded subprogram?

An **overloaded subprogram** is a subprogram that has the same name as another subprogram in the same referencing environment.

36. Define static binding and dynamic binding?

[explain you answer with examples for each one]

Static: The binding which can be resolved at compile time by compiler, this method cannot be overridden and the type of the class is determined at the compile time.

Dynamic Binding: When compiler is not able to resolve the call/binding at compile time, The type of object is determined at the run time.

37. Explain why dynamic type bindings is closely related to implicit heap-dynamic variables, using the following code example:

var a; …

a = “Hello”;

… a = 123.456;

The variable a is bounded at runtime, in this case, firstly a is “Hello” and then it changes to “123.456”. And implicit heap-dynamic variables are bounded to heap storage only when they are assigned values, which is matched with dynamic type bindings

38. Describe the case when a variable does not have a name

Explicit heap-dynamic variables are nameless (abstract) memory cells. They often do not have identifiers associated with them. They can be referenced only by pointer or reference type variables.

39. Describe the ways that aliases can occur with pass-by-reference parameters

Aliases are two or more variables bound to the same memory location. An example: int a=0; int &a1=a; In this case, a and a1 are aliases

40. What is a keyword?

A keyword is a word that is special only in certain contexts.

41. What is a static ancestor of a subprogram? Show an example (2 marks)

The static ancestor of a subprogram is its static parent and the static parent derived from it, until the largest one contains all the subprogram. Example:

function() { //it is static ancestor of sub2

function\_sub1() {

int x=1;

function\_sub2(); }

function\_sub2() {

int y=x; } }

42. Define scope and lifetime.

Scope: The range of statements where the variable is visible.

Lifetime: The time when the variable is bound to a specific memory location.

**CHAPTERWISE**

**Chapter 2 - Preliminaries**

Review Questions

1. Why is it useful for a programmer to have the ability to learn new languages, even though he or she may have a good knowledge of a number of programming languages?

Increased capacity to express ideas

Improved background for choosing appropriate languages.

Increased ability to learn new languages

Better understanding of the significance of implementation

Better use of languages that are already known.

Overall advancement of computing

1. Why is it essential to choose an appropriate programming language for a specific software solution?

Certain kinds of program bugs can be found and fixed only by a programmer who knows some related implementation details. Another benefit of understanding implementation issues is that it allows us to visualize how a computer executes various language constructs. In some cases, some knowledge of implementation issues provides hints about the relative efficiency of alternative constructs that may be chosen for a program. For example, programmers who know little about the complexity of the implementation of subprogram calls often do not realize that a small subprogram that is frequently called can be a highly inefficient design choice.

1. Which programming language for scientific applications was the first to be used successfully?

**The first language for scientific applications was Fortran.**

1. Which is the first successful high-level programming language for business?

**The first successful high-level language for business was COBOL** (ISO/IEC, 2002)

1. In which programming language were most of the AI applications developed prior to 1990?

**Most AI applications developed prior to 1990 were written in Lisp** or one of its close relatives.

1. Which is the most popular markup language for Web development?

**HTML**

1. Why is a list of programming language evaluation criteria for the development of software controversial?

Set of evaluation criteria. Such a list of criteria is necessarily controversial, because it is difficult to get even two computer scientists to agree on the value of some given language characteristic relative to others. Some of these characteristics are broad and somewhat vague, such as writability, whereas others are specific language constructs, such as exception handling.

1. How does the overall simplicity of a programming language affect its readability?

The overall simplicity of a programming language strongly affects its readability. A language with a large number of basic constructs is more difficult to learn than one with a smaller number. Programmers who must use a large language often learn a subset of the language and ignore its other features. Readability problems occur whenever the program’s author has learned a different subset from that subset with which the reader is familiar.

1. Why is the VAX instruction design orthogonal?

**The VAX instruction design is orthogonal in that a single instruction can use either registers or memory cells as the operands**.

1. Why does too much orthogonality cause problems?

Too much orthogonality can also cause problems. Perhaps the most orthogonal programming language is ALGOL 68 (van Wijngaarden et al., 1969). Every language construct in ALGOL 68 has a type, and there are no restrictions on those types. In addition, most constructs produce values. This combinational freedom allows extremely complex constructs. Extreme form of orthogonality can lead to unnecessary complexity. Furthermore, because languages require a large number of primitives, a high degree of orthogonality results in an explosion of combinations. So, even if the combinations are simple, their sheer numbers lead to complexity.

1. Explain how “writability” is used as a measure of how easily a language can be used to create programs.

**Writability** is a measure of how easily a language can be used to create programs for a chosen problem domain. Most of the language characteristics that affect readability also affect writability. This follows directly from the fact that the process of writing a program requires the programmer frequently to reread the part of the program that is already written.

1. Why is too much orthogonality a detriment to “writability”?

**too much orthogonality can be a detriment to writability**. Errors in programs can go undetected when nearly any combination of primitives is legal. This can lead to code absurdities that cannot be discovered by the compiler.

1. Give an example of expressivity in a language.

For example, in C, the notation count++ is more convenient and shorter than

count = count + 1

1. What is type checking?

**Type checking** is simply testing for type errors in a given program, either by the compiler or during program execution. Type checking is an important factor in language reliability. Because run-time type checking is expensive, compile-time type checking is more desirable. Furthermore, the earlier errors in programs are detected, the less expensive it is to make the required repairs.

1. Give an example of how the failure to type check, at either compile time or run time, can lead to countless program errors.

One example of how failure to type check, at either compile time or run time, has led to countless program errors is the use of subprogram parameters in the original C language (Kernighan and Ritchie, 1978). In this language, the type of an actual parameter in a function call was not checked to determine whether its type matched that of the corresponding formal parameter in the function. An int type variable could be used as an actual parameter in a call to a function that expected a float type as its formal parameter, and neither the compiler nor the run-time system would detect the inconsistency. For example, because the bit string that represents the integer 23 is essentially unrelated to the bit string that represents a floating-point 23, if an integer 23 is sent to a function that expects a floating-point parameter, any uses of the parameter in the function will produce nonsense. Furthermore, such problems are often difficult to diagnose.

1. How is the total cost of a programming language calculated?

First, there is the cost of training programmers to use the language,

Second, there is the cost of writing programs in the language

Third, there is the cost of compiling programs

Fourth, the cost of executing programs

The fifth factor is the cost of the language implementation system

Sixth, there is the cost of poor reliability

The final consideration is the cost of maintaining programs

1. What is portability of a language?

**Portability** is the ease with which programs can be moved from one implementation to another.

1. What is the use of the well-definedness criterion?

**Generality** (the applicability to a wide range of applications) and **well-definedness** (the completeness and precision of the language’s official defining document) are two other criteria.

**Computer Architecture**

Most of the popular languages of the past 60 years have been designed around the prevalent computer architecture, called the von Neumann architecture, after one of its originators, John von Neumann. These languages are called **imperative** languages. In a von Neumann computer, both data and programs are stored in the same memory. The central processing unit (CPU), which executes instructions, is separate from the memory. Therefore, instructions and data must be transmitted, or piped, from memory to the CPU. Results of operations in the CPU must be moved back to memory.

Because of the von Neumann architecture, the central features of imperative languages are variables, which model the memory cells; assignment statements, which are based on the piping operation; and the iterative form of repetition, which is the most efficient way to implement repetition on this architecture. Operands in expressions are piped from memory to the CPU, and the result of evaluating the expression is piped back to the memory cell represented by the left side of the assignment. Iteration is fast on von Neumann computers because instructions are stored in adjacent cells of memory and repeating the execution of a section of code requires only a branch instruction.

1. How does the execution of a machine code program on a von Neumann architecture computer occur?

The execution of a machine code program on a von Neumann architecture computer occurs in a process called the **fetch-execute cycle**. As stated earlier, programs reside in memory but are executed in the CPU. Each instruction to be executed must be moved from memory to the processor. The address of the next instruction to be executed is maintained in a register called the program counter. The fetch- execute cycle can be simply described by the following algorithm:

*initialize the program counter*

*repeat forever*

*fetch the instruction pointed to by the program counter*

*increment the program counter to point at the next instruction*

*decode the instruction*

*execute the instruction*

*end repeat*

1. What two programming language deficiencies were discovered as a result of the research in software development in the 1970s?

The primary programming **language deficiencies that were discovered were incompleteness of type checking and inadequacy of control statements** (requiring the extensive use of gotos).

1. What are the three fundamental features of an object-oriented programming language?

The latest step in the evolution of data-oriented software development, which began in the early 1980s, is object- oriented design. **Fundamental features of Object-oriented methodology:**

* Data abstraction, which encapsulates processing with data objects and controls access to data, and adds inheritance and dynamic method binding.
* Inheritance is a powerful concept that greatly enhances the potential reuse of existing software, thereby providing the possibility of significant increases in software development productivity. This is an important factor in the increase in popularity of object-oriented languages.
* Dynamic (run- time) method binding allows more flexible use of inheritance

1. What language was the first to support the three fundamental features of object-oriented programming?

Object- oriented programming developed along with a language that supported its concepts: **Smalltalk** (Goldberg and Robson, 1989).

1. What is an example of two language design criteria that are in direct conflict with each other?

The programming language evaluation criteria provides a framework for language design. Unfortunately, that framework is self-contradictory. **Two criteria that conflict are reliability and cost of execution.**

For example, the Java language definition demands that all references to array elements be checked to ensure that the index or indices are in their legal ranges. This step adds a great deal to the cost of execution of Java programs that contain large numbers of references to array elements. C does not require index range checking, so C programs execute faster than semantically equivalent Java programs, although Java programs are more reliable. The designers of Java traded execution efficiency for reliability.

Example 2. **The conflict between writability and reliability is a common one in language design.** The pointers of C++ can be manipulated in a variety of ways, which supports highly flexible addressing of data. Because of the potential reliability problems with pointers, they are not included in Java.

1. What are the three general methods of implementing a programming language?

Programming languages can be implemented by any of three general methods.

* **Compiler implementation**
* **Pure interpretation**
* **Hybrid Implementation Systems**

1. Which produces faster program execution, a compiler or a pure interpreter?

(Compiler is faster) The pure interpreter is 10-100 times slower than in compiled system, because of the decoding of the high-level language statements, which are far more complex than machine language instructions

1. What role does the symbol table play in a compiler?

**The symbol table** serves as a database for the compilation process. The primary contents of the symbol table are the type and attribute information of each user-defined name in the program. This information is placed in the symbol table by the lexical and syntax analyzers and is used by the semantic analyzer and the code generator.

1. What does a linker do?

The linking operation connects the user program to the system programs by placing the addresses of the entry points of the system programs in the calls to them in the user program. The user and system code together are sometimes called a load module, or executable image. The process of collecting system programs and linking them to user programs is called linking and loading, or sometimes just linking. It is accomplished by a systems program called a **linker**

1. Why is the von Neumann bottleneck important?

The speed of the connection between a computer’s memory and its processor often determines the speed of the computer, because instructions often can be executed faster than they can be moved to the processor for execution. This connection is called the **von Neumann bottleneck**; it is the primary limiting factor in the speed of von Neumann architecture computers. The von Neumann bottleneck has been one of the primary motivations for the research and development of parallel computers.

1. What are the advantages in implementing a language with a pure interpreter?

Pure interpretation has the advantage of allowing easy implementation of many source-level debugging operations, because all run-time error messages can refer to source-level units. For example, if an array index is found to be out of range, the error message can easily indicate the source line of the error and the name of the array. On the other hand, this method has the serious disadvantage that execution is 10 to 100 times slower than in compiled systems.

**Chapter 3 - Describing Syntax and Semantics**

Review Questions

1. Define *lexeme* and *token*.

**Lexemes** - lowest- level syntactic units (small units). The lexemes of a programming language include its numeric literals, operators, and special words, among others. One can think of programs as strings of lexemes rather than of characters. Lexemes are partitioned into groups— for example, the names of variables, methods, classes, and so forth in a programming language form a group called identifiers.

Each lexeme group is represented by a name, or token. So, a **token** of a language is a category of its lexemes. For example, an identifier is a token that can have lexemes, or instances, such as *sum* and *total*. In some cases, a token has only a single possible lexeme. For example, the token for the arithmetic operator symbol *+* has just one possible lexeme.

answer 2. logical groupings are named **lexemes**, and the internal codes for categories of these groupings are named **tokens**.

1. How are programming languages formally defined?

Programming languages can be formally defined in two distinct ways: by **recognition** and by **generation**.

1. In which form is the programming language syntax commonly described?

This revised method of syntax description became known as **Backus-Naur Form**, or simply **BNF**.

1. What is a metalanguage?

A **metalanguage** is a language that is used to describe another language. BNF is a metalanguage for programming languages.

1. What is a derivation in the context of grammar?

A grammar is a generative device for defining languages. The sentences of the language are generated through a sequence of applications of the rules, beginning with a special nonterminal of the grammar called the **start symbol**. This sequence of rule applications is called a **derivation**.

1. What is an ambiguous grammar?

A grammar that generates a sentential form for which there are two or more distinct parse trees is said to be **ambiguous**.

1. What is a left-recursive grammar?

When a grammar rule has its LHS also appearing at the beginning of its RHS, the rule is said to be **left recursive**. This left recursion specifies left associativity.

1. Explain the use of metasymbols in EBNFs.

The brackets, braces, and parentheses in the EBNF extensions are **metasymbols**, which means they are notational tools and not terminal symbols in the syntactic entities they help describe. In cases where these metasymbols are also terminal symbols in the language being described, the instances that are terminal symbols can be underlined or quoted

1. What is the purpose of a predicate function?

**Predicate functions**, which state the *static semantic rules* of the language, are associated with grammar rules.

1. When can the parse tree be called fully attributed?

A parse tree of an attribute grammar is the parse tree based on its underlying BNF grammar, with a possibly empty set of attribute values attached to each node. If all the attribute values in a parse tree have been computed, the tree is said to be **fully attributed**.

1. How is the order of evaluation of attributes determined for the trees of a given attribute grammar?

Determining attribute evaluation order for the general case of an attribute grammar is a complex problem, requiring the construction of a dependency graph to show all attribute dependencies.

1. What is the use of intrinsic attributes?

**Intrinsic attributes** are synthesized attributes of leaf nodes whose values are determined outside the parse tree. For example, the type of an instance of a variable in a program could come from the symbol table, which is used to store variable names and their types. The contents of the symbol table are set based on earlier declaration statements. Initially, assuming that an unattributed parse tree has been constructed and that attribute values are needed, the only attributes with values are the intrinsic attributes of the leaf nodes. Given the intrinsic attribute values on a parse tree, the semantic functions can be used to compute the remaining attribute values

1. What is meant by decorating a parse tree?

the process of computing the attribute values of a parse tree, which is sometimes called **decorating the parse tree**

1. Why can machine languages not be used to define statements in operational semantics?

Machine languages and real computers are not used for formal operational semantics.There are several problems with using this approach for complete formal semantics descriptions.

* First, the individual steps in the execution of machine language and the resulting changes to the state of the machine are too small and too numerous.

* Second, the storage of a real computer is too large and complex. There are usually several levels of memory devices, as well as connections to enumerable other computers and memory devices through networks.

1. Describe the two levels of uses of operational semantics.

There are different levels of uses of operational semantics.

* At the highest level, the interest is in the final result of the execution of a complete program. This is sometimes called **natural operational semantics**.
* At the lowest level, operational semantics can be used to determine the precise meaning of a program through an examination of the complete sequence of state changes that occur when the program is executed. This use is sometimes called **structural operational semantics**.

1. In denotational semantics, what are the syntactic and semantic domains?

*The domain* is the collection of values that are legitimate parameters to the function. In denotational semantics, the domain is called the **syntactic domain**, because it is syntactic structures that are mapped.

*The range* is the collection of objects to which the parameters are mapped. In denotational semantics, the range is called the **semantic domain**.

1. What is stored in the state of a program for denotational semantics?

Denotational semantics is defined in terms of only the values of all of the program’s variables. So, denotational semantics uses the state of the program to describe meaning, whereas operational semantics uses the state of a machine. The key difference between operational semantics and denotational semantics is that state changes in operational semantics are defined by coded algorithms, written in some programming language, whereas in denotational semantics, state changes are defined by mathematical functions

1. What is an assertion in axiomatic semantics?

The logical expressions used in axiomatic semantics are called predicates, or **assertions**.

1. What two things must be defined for each language entity in order to construct a denotational description of the language?

The process of constructing a denotational semantics specification for a programming language requires one to define for each language entity both a mathematical object and a function that maps instances of that language entity onto instances of the mathematical object.

1. Which part of an inference rule is the antecedent?

An **inference rule** is a method of inferring the truth of one assertion on the basis of the values of other assertions. The general form of an inference rule is as follows:

C:\Users\User\AppData\Local\Temp\ksohtml29292\wps1.jpg

This rule states that if S1, S2, . . . , and Sn are true, then the truth of S can be inferred.

The top part of an inference rule is called its **antecedent**; the bottom part is called its **consequent**.

1. What is the weakest precondition?

The weakest precondition for a sequence of statements cannot be described by an axiom, because the precondition depends on the particular kinds of statements in the sequence. In this case, the precondition can only be described with an inference rule

1. What is an inference rule?

An **inference rule** is a method of inferring the truth of one assertion on the basis of the values of other assertions. The general form of an inference rule is as follows:

C:\Users\User\AppData\Local\Temp\ksohtml29292\wps2.jpg

This rule states that if S1, S2, . . . , and Sn are true, then the truth of S can be inferred.

1. Give an example of a logical pretest loop.

While loop in C

1. On what branch of mathematics is denotational semantics based?

It is based on mathematics and logic (logical branch of mathematics). It is solidly based on recursive function theory.

1. What is the problem with using a software pure interpreter for operational semantics?
2. Explain what the preconditions and postconditions of a given statement mean in axiomatic semantics.

* An assertion immediately preceding a program statement describes the constraints on the program variables at that point in the program. These assertions are called the **precondition**.
* An assertion immediately following a statement describes the new constraints on those variables (and possibly others) after execution of the statement. These assertions are called the **postcondition**.

1. What is the difference between total correctness and partial correctness with regard to loop termination?

If loop termination can be shown, the axiomatic description of the loop is called **total correctness**. If the other conditions can be met but termination is not guaranteed, it is called **partial correctness**.

1. When is a function called a predicate transformer?

A function is often called a **predicate transformer**, because it takes a predicate, or assertion, as a parameter and returns another predicate.

1. In what fundamental way do operational semantics and denotational semantics differ?

Denotational semantics is related to operational semantics. In operational semantics, programming language constructs are translated into simpler programming language constructs, which become the basis of the meaning of the construct. In denotational semantics, programming language constructs are mapped to mathematical objects, either sets or, more often, functions. However, unlike operational semantics, denotational semantics does not model the step-by-step computational processing of programs.

**Chapter 4 - Lexical and Syntax Analysis**

Review Questions

1. What are the reasons why using BNF is advantageous over using an informal syntax description?

Using BNF, as opposed to using some informal syntax description, has at least three compelling advantages.

* BNF descriptions of the syntax of programs are clear and concise, both for humans and for software systems that use them.
* The BNF description can be used as the direct basis for the syntax analyzer.
* Implementations based on BNF are relatively easy to maintain because of their modularity

1. How does a lexical analyzer serve as the front end of a syntax analyzer?

A **lexical analyzer serves as the front end of a syntax analyzer**. Technically, lexical analysis is a part of syntax analysis. A lexical analyzer performs syntax analysis at the lowest level of program structure. An input program appears to a compiler as a single string of characters. The lexical analyzer collects characters into logical groupings and assigns internal codes to the groupings according to their structure.

1. Define finite automata and regular grammar.

State diagrams of the form used for lexical analyzers are representations of a class of mathematical machines called **finite automata**. Finite automata can be designed to recognize members of a class of languages called regular languages.

**Regular grammars** are generative devices for regular languages. The tokens of a programming language are a regular language, and a lexical analyzer is a finite automaton.

1. How can you construct a lexical analyzer with a state diagram?

We now illustrate lexical-analyzer construction with a state diagram and the code that implements it.

The state diagram could simply include states and transitions for each and every token pattern. However, that approach results in a very large and complex diagram, because every node in the state diagram would need a transition for every character in the character set of the language being analyzed. We therefore consider ways to simplify it.

1. Describe briefly the three approaches to building a lexical analyzer.

There are **three approaches to building a lexical analyzer**:

1. Write a formal description of the token patterns of the language using a descriptive language related to regular expressions. These descriptions are used as input to a software tool that automatically generates a lexical analyzer.
2. Design a state transition diagram that describes the token patterns of the language and write a program that implements the diagram.
3. Design a state transition diagram that describes the token patterns of the language and hand-construct a table-driven implementation of the state diagram.
4. What are the different grammar symbols for formal languages?

Different grammar symbols for formal languages:

* **Terminal symbols**—lowercase letters at the beginning of the alphabet (a, b, . . .)
* **Nonterminal symbols**—uppercase letters at the beginning of the alphabet (A, B, . . .)
* **Terminals** or **nonterminals**—uppercase letters at the end of the alphabet (W, X, Y, Z)
* **Strings of terminals**—lowercase letters at the end of the alphabet (w, x, y, z)
* **Mixed strings** (terminals and/or nonterminals)—lowercase Greek letters (α, β, δ, γ)

1. Why are character classes used, rather than individual characters, for the letter and digit transitions of a state diagram for a lexical analyzer?

Names and reserved words in programs have similar patterns. Although it is possible to build a state diagram to recognize every specific reserved word of a programming language, that would result in a prohibitively large state diagram. It is much simpler and faster to have the lexical analyzer recognize names and reserved words with the same pattern and use a lookup in a table of reserved words to determine which names are reserved words. Using this approach considers reserved words to be exceptions in the names token category.

1. What are the two distinct goals of syntax analysis?

The syntax analyzer must check the input program to determine whether it is syntactically correct.

The second goal of syntax analysis is to produce a complete parse tree,

1. Describe the complexity of parsing algorithms.

Parsing algorithms that work for any unambiguous grammar are complicated and inefficient.

**In fact, the complexity of such algorithms is O(n^3)**, which means the amount of time they take is on the order of the cube of the length of the string to be parsed.

1. Describe the recursive-descent parser.

A **recursive-descent parser** is a coded version of a syntax analyzer based directly on the BNF description of the syntax of language.

1. What do the two Ls in LL algorithm specify?

Both of these, which are called **LL algorithms**, are equally powerful, meaning they work on the same subset of all context-free grammars.

* **The first L** in LL specifies a left-to-right scan of the input.
* **The second L** specifies that a leftmost derivation is generated.

1. State and explain the convention followed for writing a recursive-descent parsing subprogram.

Recursive-descent parsing subprograms are written with the convention that each one leaves the next token of input in *nextToken*. So, whenever a parsing function begins, it assumes that *nextToken* has the code for the leftmost token of the input that has not yet been used in the parsing process.

1. Why are named constants used, rather than numbers, for token codes?

Although **tokens** are usually represented as integer values, for the sake of readability of lexical and syntax analyzers, they are often referenced through named constants.

1. Describe how a recursive-descent parsing subprogram is written for a rule with a single RHS.

A recursive-descent subprogram for a rule with a single RHS is relatively simple.

For each terminal symbol in the RHS, that terminal symbol is compared with *nextToken*. If they do not match, it is a syntax error. If they match, the lexical analyzer is called to get the next input token. For each nonterminal, the parsing subprogram for that nonterminal is called.

1. Explain the two grammar characteristics that prohibit them from being used as the basis for a top-down parser.

One simple grammar characteristic that causes a catastrophic problem for LL parsers is **left recursion**.

For example, consider the following rule:

*A -> A + B*

A recursive-descent parser subprogram for A immediately calls itself to parse the first symbol in its RHS. That activation of the A parser subprogram then immediately calls itself again, and again, and so forth. It is easy to see that this leads nowhere (except to stack overflow).

The left recursion in the rule *A -> A + B* is called **direct left recursion**, because it occurs in one rule**.**

1. What is direct left recursion?

The left recursion in the rule *A -> A + B* is called **direct left recursion**, because it occurs in one rule**.**

1. Describe the pairwise disjointness test.

There is a relatively simple test of a non-left recursive grammar that indicates whether this can be done, called the **pairwise disjointness test.**

1. What is the limitation of left factoring?

Left factoring cannot solve all pairwise disjointness problems of grammars.

1. What is a phrase of a right sentential form?

**Phrase**, relative to a parse tree, is the string of all of the leaves of the partial parse tree that is rooted at one particular internal node of the whole parse tree.

1. What is the difference between a simple phrase and a phrase of the right sentential form?

The definition of **phrase** uses one or more steps, while the definition of **simple phrase** uses exactly one step.

1. What is the feature of the handle of a right sentential form?

**The handle of a right sentential form is unique**.

The **handle** of any rightmost sentential form is its leftmost simple phrase.

1. What is the mathematical machine on which both top-down and bottom-up parsers are based?

A **PDA** is a very simple mathematical machine that scans strings of symbols from left to right.

1. What is the disadvantage of an LR parser?

The only **disadvantage** of **LR parsing** is that it is difficult to produce by hand the parsing table for a given grammar for a complete programming language.

1. Why is a bottom-up parser often called a shift-reduce algorithm?

Bottom-up parsers are often called **shift-reduce algorithms**, because shift and reduce are the two most common actions they specify.

1. Describe the purpose of a parse stack in an LR parser.

Each situation could be represented by a state and stored in the **parse stack**, one state symbol for each grammar symbol on the stack. At the top of the stack would always be a state symbol, which represented the relevant information from the entire history of the parse, up to the current time.

1. Describe the properties of the variations on the canonical LR table construction process.

These are characterized by two properties:

1. They require far less computer resources to produce the required parsing table than the canonical LR algorithm.
2. They work on smaller classes of grammars than the canonical LR algorithm.
3. Why is every parser for a programming language a pushdown automaton?

Every parser for a programming language is a **pushdown automaton (PDA)**, because a PDA is a recognizer for a context-free language.

**Chapter 5 - Names, Bindings, and Scopes**

Review Questions

* + - 1. What is a reserved word?

A **reserved word** is a special word of a programming language that cannot be used as a name.

2.In Java and C#, how long can a name be?

**Names in Java and C# have no length limit, and all characters in them are significant**

1. What is the address of a variable?

The **address of a variable** is the machine memory address with which it is associated.

1. How can a variable be characterized?

A **variable can be characterized** as a sextuple of attributes: (name, address, value, type, lifetime, and scope).

1. What is type inference? Give an example.

Another kind of implicit type declarations uses context. This is sometimes called **type inference**. In the simpler case, the context is the type of the value assigned to the variable in a declaration statement.

For example, in C# a ***var*** declaration of a variable must include an initial value, whose type is taken as the type of the variable.

1. What is deallocation of a memory cell?

**Deallocation** is the process of placing a memory cell that has been unbound from a variable back into the pool of available memory.

1. After language design and implementation, what are the four times bindings can take place in a program?

Bindings can take place at language design time, language implementation time, **compile time, load time, link time, or run time**.

1. What is the lifetime of a variable?

The **lifetime** of a variable is the time during which the variable is bound to a specific memory location. So, the lifetime of a variable begins when it is bound to a specific cell and ends when it is unbound from that cell.

**9** . What is the use of a stack-dynamic variable?

* The use of stack-dynamic variables are as follows: To be useful, at least in most cases, recursive subprograms require some form of dynamic local storage so that each active copy of the recursive subprogram has its own version of the local variables. These needs are conveniently met by stack-dynamic variables. Even in the absence of recursion, having stack-dynamic local storage for subprograms is not without merit, because all subprograms share the same memory space for their locals.

10.What are the advantages and disadvantages of dynamic type binding?

* The primary **advantage** of dynamic binding of variables to types is that it provides more programming flexibility.
* There are two **disadvantages** to dynamic type binding.

First, it causes programs to be less reliable, because the error-detection capability of the compiler is diminished relative to a compiler for a language with static type bindings

The greatest disadvantage of dynamic type binding is cost.

**11**. Define *static*, *stack-dynamic*, *explicit heap-dynamic*, and *implicit heap-dynamic variables*. What are their advantages and disadvantages?

**Static variables** are those that are bound to memory cells before program execution begins and remain bound to those same memory cells until program execution terminates.

* One advantage of static variables is efficiency. All addressing of static variables can be direct; other kinds of variables often require indirect addressing, which is slower. Also, no run- time overhead is incurred for allocation and deallocation of static variables
* One disadvantage of static binding to storage is reduced flexibility; in particular, a language that has only static variables cannot support recursive subprograms. Another disadvantage is that storage cannot be shared among variables. For example, suppose a program has two subprograms, both of which require large arrays. Furthermore, suppose that the two subprograms are never active at the same time. If the arrays are static, they cannot share the same storage for their arrays.

**Stack-dynamic variables**

**Stack-dynamic variables** are those whose storage bindings are created when their declaration statements are elaborated, but whose types are statically bound.

**Elaboration** of such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached. Therefore, elaboration occurs during run time.

As their name indicates, stack- dynamic variables are allocated from the run-time stack.

* The advantages of stack-dynamic variables are as follows: To be useful, at least in most cases, recursive subprograms require some form of dynamic local storage so that each active copy of the recursive subprogram has its own version of the local variables. These needs are conveniently met by stack-dynamic variables. Even in the absence of recursion, having stack-dynamic local storage for subprograms is not without merit, because all subprograms share the same memory space for their locals.
* The disadvantages of stack-dynamic variables are the run- time overhead of allocation and deallocation, possibly slower accesses because indirect addressing is required, and the fact that subprograms cannot be history sensitive. The time required to allocate and deallocate stack-dynamic variables is not significant, because all of the stack-dynamic variables that are declared at the beginning of a subprogram are allocated and deallocated together, rather than by separate operations.

**Explicit Heap-Dynamic Variables**

**Explicit heap- dynamic variables** are nameless (abstract) memory cells that are allocated and deallocated by explicit run-time instructions written by the programmer. These variables, which are allocated from and deallocated to the heap, can only be referenced through pointer or reference variables. The heap is a collection of storage cells whose organization is highly disorganized due to the unpredictability of its use.

Page 237

* Explicit heap-dynamic variables are often used to construct dynamic structures, such as linked lists and trees, that need to grow and/or shrink during execution. Such structures can be built conveniently using pointers or references and explicit heap-dynamic variables.
* The disadvantages of explicit heap-dynamic variables are the difficulty of using pointer and reference variables correctly, the cost of references to the variables, and the complexity of the required storage management implementation. This is essentially the problem of heap management, which is costly and complicated.

**Implicit Heap-Dynamic Variables Implicit heap-dynamic variables** are bound to heap storage only when they are assigned values. In fact, all their attributes are bound every time they are assigned.

Page 238

* The advantage of such variables is that they have the highest degree of flexibility, allowing highly generic code to be written.

One disadvantage of implicit heap- dynamic variables is the run- time overhead of maintaining all the dynamic attributes, which could include array subscript types and ranges, among others. Another disadvantage is the loss of some error detection by the compiler

**12**. What is a block-structured language?

Many languages allow new static scopes to be defined in the midst of executable code. This powerful concept allows a section of code to have its own local variables whose scope is minimized. Such variables are typically stack dynamic, so their storage is allocated when the section is entered and deallocated when the section is exited. Such a section of code is called a **block**. Blocks provide the origin of the phrase **block-structured language**.

**13**. How is a reference to a nonlocal variable in a static-scoped program connected to its definition?

When the reader of a program finds a reference to a variable, the attributes of the variable can be determined by finding the statement in which it is declared (either explicitly or implicitly).

**14**. What is the general problem with static scoping?

in most cases **it allows more access to both variables and subprograms than is necessary**.

**15**. What is the referencing environment of a statement?

**The referencing environment of a statement** is the collection of all variables that are visible in the statement

**16**. What is a static ancestor of a subprogram? What is a dynamic ancestor of a subprogram?

The static parent of subprogram, and its static parent, and so forth up to and including the largest enclosing subprogram, are called the **static ancestors** .

Dynamic ancestors of a subprogram are all the procedures called before it during the execution of a program

**17**. What is a block?

It is the scope of local variables in a piece of code (this section of code between each pair of braces)

**18**. What is the purpose of the let constructs in functional languages?

Most functional programming languages include a construct that is related to the blocks of the imperative languages, usually named *let*. These constructs have two parts:

* The first of which is to bind names to values, usually specified as expressions.
* The second part is an expression that uses the names defined in the first part.

**19**. What is the difference between the names defined in an ML let construct from the variables declared in a C block?

In **ML**, the form of a let construct is as follows:

***let***

***val*** *name1 = expression1*

*. . .*

***val*** *namen = expressionn*

***in***

*expression*

***end****;*

Each ***val*** statement binds a name to an expression.

The C- based languages allow any compound statement (a statement sequence surrounded by matched **braces**)

**20**. Describe the encapsulation of an F# let inside a function and outside all functions.

The general form of a ***let*** construct in **F#** is as follows:

***let*** *left\_side = expression*

The *left\_side* of ***let*** can be a name or a tuple pattern (a sequence of names separated by commas).

The scope of a name defined with let inside a function definition is from the end of the defining expression to the end of the function. The scope of let can be limited by indenting the following code, which creates a new local scope. Although any indentation will work, the convention is that the indentation is four spaces. Consider the following code:

***let*** *n1 =*

***let*** *n2 = 7*

***let*** *n3 = n2 + 3*

*n3;;*

***let*** *n4 = n3 + n1;;*

The scope of *n1* extends over all of the code. However, the scope of *n2* and *n3* ends when the indentation ends. So, the use of *n3* in the last let causes an error. The last line of the let *n1* scope is the value bound to *n1*; it could be any expression.

**21**. When can we call a subprogram active?

**A subprogram is active** if its execution has begun but has not yet terminated.

**22**. What is variable initialization?

The binding of a variable to a value at the time it is bound to storage is called **initialization**.

**Chapter 6 - Data Types**

Review Questions

* 1. What is a data type?
* A **data type** defines a collection of data values and a set of predefined operations on those values.

**2**. What are the advantages of the data structure of COBOL over the data structure of Fortran I?

in pre-90 Fortrans, linked lists and binary trees were implemented with arrays. The data structures of COBOL took the first step away from the Fortran I model by allowing programmers to specify the accuracy of decimal data values, and also by providing a structured data type for records of information.

3.What is an abstract data type?

The fundamental idea of an **abstract data type** is that the interface of a type, which is visible to the user, is separated from the representation and set of operations on values of that type, which are hidden from the user. All of the types provided by a high-level programming language are abstract data types.

1. Describe the different uses of the type system of a programming language.

There are a number of uses of the type system of a programming language.

* The most practical of these is **error detection**.
* A second use of a type system is the assistance it provides for **program modularization**. This results from the cross- module type checking that ensures the consistency of the interfaces among modules.
* Another use of a type system is **documentation**. The type declarations in a program document information about its data, which provides clues about the program’s behavior.

1. Define *descriptor* and *object*.

A ***descriptor*** is the collection of the attributes of a variable. In an implementation, a descriptor is an area of memory that stores the attributes of a variable.

We reserve ***object*** exclusively for instances of user-defined and language-defined abstract data types, rather than for the values of all program variables of predefined types.

1. What are the two most common structured data types in the imperative languages?

The two most common structured (nonscalar) data types in the imperative languages are **arrays** and **records**.

1. What mechanism is used to store negative integers in a computer?

Most computers now use a notation called **twos complement** to store negative integers

1. What are the four signed integers supported by Java?

Java includes four signed integer sizes: ***byte***, ***short***, ***int***, and ***long***.

1. Define *static*, *fixed stack-dynamic*, *fixed heap-dynamic*, and *heap-dynamic arrays*. What are the advantages of each?

A **static** **array** is one in which the subscript ranges are statically bound and storage allocation is static (done before run time).

The advantage of static arrays is efficiency: No dynamic allocation or deallocation is required.

The disadvantage is that the storage for the array is fixed for the entire execution time of the program.

A **fixed stack-dynamic array** is one in which the subscript ranges are statically bound, but the allocation is done at declaration elaboration time during execution.

The advantage of fixed stack-dynamic arrays over static arrays is space efficiency.

A large array in one subprogram can use the same space as a large array in a different subprogram, as long as both subprograms are not active at the same time. The same is true if the two arrays are in different blocks that are not active at the same time.

The disadvantage is the required allocation and deallocation time.

A **fixed heap-dynamic array** is similar to a fixed stack-dynamic array, in that the subscript ranges and the storage binding are both fixed after storage is allocated. The differences are that both the subscript ranges and storage bindings are done when the user program requests them during execution, and the storage is allocated from the heap, rather than the stack.

The advantage of fixed heap-dynamic arrays is flexibility—the array’s size always fits the problem.

The disadvantage is allocation time from the heap, which is longer than allocation time from the stack.

A **heap- dynamic array** is one in which the binding of subscript ranges and storage allocation is dynamic and can change any number of times during the array’s lifetime.

The advantage of heap-dynamic arrays over the others is flexibility: Arrays can grow and shrink during program execution as the need for space changes.

The disadvantage is that allocation and deallocation take longer and may happen many times during execution of the program.

10.What is the disadvantage of ones-complement notation?

Ones-complement notation has the disadvantage that it has two representations of zero.

11.How are floating-point types represented?

Floating-point values are represented **as fractions and exponents**, a form that is borrowed from scientific notation.

**12**. What are the different representations for floating-point values supported by Java?

Most languages include two floating- point types, often called **float** and **double**.

* The **float** **type** is the standard size, usually stored in four bytes of memory.
* The **double type** is provided for situations where larger fractional parts and/or a larger range of exponents is needed. Double- precision variables usually occupy twice as much storage as float variables and provide at least twice the number of bits of fraction.

**13**. What languages support complex data type?

Some programming languages support a **complex data type**—for example, Fortran and Python.

**14**. What is the disadvantage of a decimal type?

The **disadvantages** of decimal types are that the range of values is restricted because no exponents are allowed, and their representation in memory is mildly wasteful.

**15**. Define precision and range.

**Precision** is the accuracy of the fractional part of a value, measured as the number of bits.

**Range** is a combination of the range of fractions and, more important, the range of exponents.

**16**. What is the use of a Boolean type?

Boolean types are often used to represent switches or flags in programs.

**17**. What are the required entries in a Java array descriptor, and when must they be stored (at compile time or run time)?

If any of the descriptor **entries are dynamically bound**, then those parts of the descriptor must be maintained at **run time**.

**18**. What is a substring reference?

* A **substring reference** is a reference to a substring of a given string. Substring references are discussed in the more general context of arrays, where the substring references are called **slices**.

**19**. What two classes support strings in Java?

In Java, strings are supported by the *String* class, whose values are constant strings, and the *StringBuffer* class, whose values are changeable and are more like arrays of single characters.

These values are specified with methods of the *StringBuffer* class

**20**. Define *limited dynamic length strings* and *dynamic length strings*.

* The second option is to allow strings to have varying length up to a declared and fixed maximum set by the variable’s definition, as exemplified by the strings in C and the C-style strings of C++. These are called **limited dynamic length strings**.
* The third option is to allow strings to have varying length with no maximum, as in JavaScript, Perl, and the standard C++ library. These are called **dynamic length strings**.

**21**. What is an immutable object of Java’s String class?

Python strings are immutable, similar to the *String* class objects of Java.

**22**. Define tuple.

A **tuple** is a data type that is similar to a record (Page 292 Q40), except that the elements are not named.

**23**. What are the design issues for enumeration types?

The design issues for enumeration types are as follows:

* Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant in the program checked?
* Are enumeration values coerced to integer?
* Are any other types coerced to an enumeration type?

**24**. In what primarily imperative language do lists serve as arrays?

Python’s arrays are called lists, although they have all the characteristics of dynamic arrays. Because the objects can be of any types, these arrays are heterogeneous.

**25**. What method is used to fetch the internal numeric value of an enumeration variable?

***ordinal*** **method**

**26**. What is the difference between C# enumeration types and C++ enumeration types?

**C# enumeration** **types** are like those of C++, except that they are never coerced to integer. So, operations on enumeration types are restricted to those that make sense. Also, the range of values is restricted to that of the particular enumeration type.

27.In what way does Scheme’s CDR function modify its parameter?

The **CDR** **function** returns its parameter list minus its first element.

1. What is the mechanics of a list comprehension in Python?

**The mechanics of a list comprehension** is that a function is applied to each of the elements of a given array and a new array is constructed from the results.

1. Define *union*, *free union*, and *discriminated union*.

A **union** is a type whose variables may store different type values at different times during program execution.

The unions in C or C++ languages are called **free unions**, because programmers are allowed complete freedom from type checking in their use.

Type checking of unions requires that each union construct include a *type indicator*. Such an indicator is called a **tag**, or **discriminant**, and a union with a discriminant is called a **discriminated union**

1. Are the unions of F# discriminated?

**YES**

**31**. What are the advantages of the enumeration types of F#?

In the area of **reliability**, the enumeration types of C#, F#, and Java 5.0 provide two advantages:

* No arithmetic operations are legal on enumeration types;

this prevents adding days of the week, for example.

* No enumeration variable can be assigned a value outside its defined range.

If the colors enumeration type has 10 enumeration constants and uses 0..9 as its internal values, no number greater than 9 can be assigned to a colors type variable.

**32**. What is the design issue for associative arrays?

The only **design issue** that is specific for associative arrays is the form of references to their elements.

**33**. Why are associative arrays called hashes in Perl?

In Perl, associative arrays are called **hashes**

**34**. What are generic arrays in Java?

C# and Java 5.0 provide **generic arrays**, that is, arrays whose elements are references to objects, through their class libraries.

**35**. Why are reference variables in C++ better than pointers for formal parameters?

When used as formal parameters in function definitions, **C++ reference types provide for two-way communication between the caller function and the called function**. This is not possible with nonpointer primitive parameter types, because **C++ parameters are passed by value**.

**36**. What advantages do Java and C# reference type variables have over the pointers in other languages?

Unlike C++ reference variables, **Java reference variables can be assigned to refer to different class instances**; they are not constants. All Java class instances are referenced by reference variables.

Because Java class instances are implicitly deallocated (there is no explicit deallocation operator), **there cannot be dangling references in Java**.

**C# includes both the references of Java and the pointers of C++.** However, the use of pointers is strongly discouraged. In fact, any subprogram that uses pointers must include the ***unsafe*** modifier.

**37**. Describe the lazy and eager approaches to reclaiming garbage.

The two most common traditional techniques are:

**Mark-sweep**, in which reclamation occurs only when the list of available space becomes empty (sometimes called the **lazy approach**).

**Reference counters**, in which reclamation is incremental and is done when inaccessible cells are created (sometimes called the **eager approach**).

**38**. Why wouldn’t arithmetic on Java and C# references make sense?

A **reference type variable** is similar to a pointer, with one important and fundamental difference:

A pointer refers to an address in memory, while a reference refers to an object or a value in memory. As a result, although it is natural to perform arithmetic on addresses, **it is not sensible to do arithmetic on references**.

**39**. What is the data type of the subscript of an array in Java?

**The type of the subscripts is often integer**.

**40**. Define *record*.

A **record** is an aggregate of data elements in which the individual elements are identified by names and accessed through offsets from the beginning of the structure.

**41**. Define strongly typed.

A programming language is **strongly typed** if type errors are always detected. This requires that the types of all operands can be determined, either at compile time or at run time.

**42**. Why is Java not strongly typed?

**Java and C#**, although they are based on C++, **are nearly strongly typed**. Types can be explicitly cast, which **could result in a type error**. However, there are no implicit ways type errors can go undetected.

**43**. What is the syntax of array references?

The syntax of array references is fairly universal: The array name is followed by the list of subscripts, which is surrounded by either parentheses or brackets.

**44**. What languages have no type coercions?

**ML and F#**

**45**. Why are the unions in C and C++ called free unions?

* **C** and **C++** provide union constructs in which there is no language support for type checking. In C and C++, the union construct is used to specify union structures.

The unions in these languages are called **free unions**, because programmers are allowed complete freedom from type checking in their use.

**46**. What is name type equivalence?

**Name type equivalence** means that two variables have equivalent types if they are defined either in the same declaration or in declarations that use the same type name.

**47**. What is a dangling pointer?

A **dangling pointer**, or **dangling reference**, is a pointer that contains the address of a heap-dynamic variable that has been deallocated.

**48**. What is the primary advantage of name type equivalence?

**Advantage: Name type equivalence** is easy to implement**.**

**49**. What is the primary disadvantage to structure type equivalence?

**Disadvantage of Structure type equivalence:** it is more difficult to implement. Under structure type equivalence, the entire structures of the two types must be compared. This comparison is not always simple.

**50**. For what types does C use structure type equivalence? ?

**nonscalar types**

**51**. What set operation models C’s *struct* data type?

**Cartesian product**

**Chapter 7 - Expressions and Assignment Statements**

Review Questions

1.Define *operator precedence* and *operator* *associativity?*

The **operator precedence rules** for expression evaluation partially define the order in which the operators of different precedence levels are evaluated. The operator precedence rules for expressions are based on the hierarchy of operator priorities, as seen by the language designer.

When an expression contains two adjacent occurrences of operators with the same level of precedence, the question of which operator is evaluated first is answered by the **associativity rules** of the language. An **operator** can have either **left** or **right** **associativity**, meaning that when there are two adjacent operators with the same precedence, the left operator is evaluated first or the right operator is evaluated first, respectively.

2.What is a *unary* operator?

An operator can be **unary**, meaning it has a single operand

3.What is an *infix* operator?

In most programming languages, binary operators are **infix**, which means they appear between their operands.

**4**. What operator usually has left associativity?

**Addition ‘+’**

**5**. When do we call operators “adjacent”?

We call operators “**adjacent**” if they are separated by a single operand.

**6**. What associativity rules are used by Java?

**Left** **associativity**

**7**. What is the difference between the way exponentiation operators are implemented in Fortran and Ruby?

Exponentiation in Fortran and Ruby is right associative

8.How do parentheses affect the precedence rule?

A parenthesized part of an expression has precedence over its adjacent unparenthesized parts.

9.How is a Lisp statement declared?

**all arithmetic and logic operations in Lisp are performed by subprograms. But in Lisp, the subprograms must be explicitly called.**

**10**. Give a solution to the problem of operand evaluation order and side effects.

There are two possible solutions to the problem of operand evaluation order and side effects.

1. First, the language designer could disallow function evaluation from affecting the value of expressions by simply disallowing functional side effects.
2. Second, the language definition could state that operands in expressions are to be evaluated in a particular order and demand that implementors guarantee that order.

11.What is an overloaded operator?

This multiple use of an operator is called **operator overloading**.

**12**. Define *narrowing* and *widening* *conversions*

* A **narrowing conversion** converts a value to a type that cannot store even approximations of all of the values of the original type.
* A **widening** **conversion** converts a value to a type that can include at least approximations of all of the values of the original type.

**13**. In JavaScript, what is the difference between == and ===?

*"7" == 7*

is true in JavaScript, because when a string and a number are the operands of a relational operator, the string is coerced to a number. However,

*"7" === 7*

is false, because no coercion is done on the operands of this operator.

**14**. What is a mixed-mode expression?

Arithmetic expressions in which operator can have operands of different types are called **mixed-mode expressions**.

**15**. How is referential transparency related to functional side effects?

The connection of referential transparency and functional side effects is illustrated by the following example:

*result1 = (fun(a) + b) / (fun(a) - c);*

*temp = fun(a);*

*result2 = (temp + b) / (temp - c);*

If the function *fun* has no side effects, *result1* and *result2* will be equal, because the expressions assigned to them are equivalent.

However, suppose *fun* has the side effect of adding *1* to either *b* or *c*. Then *result1* would not be equal to *result2*.

**16**. What are the advantages of referential transparency?

There are several **advantages** to **referentially transparent programs**.

The most important of these is that the semantics of such programs is much easier to understand than the semantics of programs that are not referentially transparent. Being referentially transparent makes a function equivalent to a mathematical function, in terms of ease of understanding.

**17**. How does operand evaluation order interact with functional side effects?

**If the language does not allow functional side effects then the order of evaluating the operands has no effects on the value of the expression**

**18**. What is *short-circuit evaluation*?

A **short-circuit evaluation** of an expression is one in which the result is determined without evaluating all of the operands and/or operators.

**19**. Name a language that always does short-circuit evaluation of Boolean expressions. Name one that never does it.

Name a language **that always does short-circuit evaluation** of Boolean expressions: **Java**

Name a language **that never does short-circuit evaluation** of Boolean expressions: ??

In the **C-based languages**

**20**. How does C support relational and Boolean expressions?

C support relational and Boolean expression by using the sign of >, >=, <, <= and == to express the relational and uses the variable type of Boolean to assign true or false or can be represented as 1 for true and 0 for false.

**21**. What is the purpose of a compound assignment operator?

A **compound assignment operator** is a shorthand method of specifying a commonly needed form of assignment

**22**. What is the associativity of C’s unary arithmetic operators?

**Right to left**

**23**. What is one possible disadvantage of treating the assignment operator as if it were an arithmetic operator?

The disadvantage of allowing assignment statements to be operands in expressions is that it provides yet another kind of expression side effect. This type of side effect can lead to expressions that are difficult to read and understand.

**24**. What two languages include multiple assignments?

**Perl and Ruby**

**25**. What mixed-mode assignments are allowed in Java?

In a clear departure from C++, **Java** and C# **allow mixed-mode assignment** **only if the required coercion is widening.**

So, an int value can be assigned to a float variable, but not vice versa. Disallowing half of the possible mixed-mode assignments is a simple but effective way to increase the reliability of Java and C#, relative to C and C++.

**26**. What mixed-mode assignments are allowed in ML?

Of course, **in functional languages (ML)**, where assignments are just used to name values, **there is no such thing as a mixed-mode assignment.**

Because error detection is reduced when **mixed-mode expressions** are allowed, F# and **ML do not allow** them. For example, they do not allow mixing of integer and floating-point operands in expressions.

**27**. What is a *cast*?

In the C-based languages, explicit type conversions are called **casts**

**Chapter 8 - Statement-Level Control Structures**

Review Questions

1.Define *selection statement*.

A **selection statement** provides the means of choosing between two or more execution paths in a program.

**2**. Mention one common misuse of the Böhm and Jacopini result.

Although a single control statement (a selectable **goto**) is minimally sufficient, a language that is designed not to include a goto needs only a small number of different control statements.

In fact, it was proven that all algorithms that can be expressed by flowcharts can be coded in a programming language with only two control statements: one for choosing between two control flow paths and one for logically controlled iterations (**Böhm and Jacopini**, 1966).

An important result of this is that the unconditional branch statement is superfluous—potentially useful but nonessential. This fact, combined with the practical problems of using unconditional branches, or gotos, led to a great deal of debate about the goto

3.What is the general form of a two-way selector?

The **general form** of a **two-way selector** is as follows:

*if control\_expression*

*then clause*

*else clause*

4.What mechanism does Python use to specify compound statements?

**indentation**

**5**. Why is a static semantics rule used instead of a syntactic entity in many languages?

A static semantics rule, rather than a syntactic entity, is used **to provide the disambiguation**.

**6**. Give an example of a language that does not allow else-less if statements.

**ML**

**7**. Under what circumstances can arithmetic expressions be used as control expressions?

In C89, **which did not have a Boolean data type, arithmetic expressions were used as control expressions.**

This can also be done in Python, C99, and C++. However, in those languages **either arithmetic or Boolean expressions can be used**. In other contemporary languages, only Boolean expressions can be used for control expressions.

9.What is the limitation of building a multiple selector from two-way selectors and gotos?

Although a **multiple selector** can be built from two-way selectors and gotos, **the** **resulting structures are cumbersome, unreliable, and difficult to write and read**.

**9**. What are the two general categories of a selection statement?

**Selection statements** fall into two general categories: **two-way** and **n-way**, or **multiple selection**.

10.Between what two language characteristics is a trade-off made when deciding whether more than one selectable segment is executed in one execution of a multiple selection statement?

**reliability** **and** **flexibility**

11.What is the role of the **default** segment in a **switch** **statement**?

If **switch** expression does not match with any case, **default statements** are executed by the program.

**12**. What functional languages do not have statements as selectors?

**ML, F#, and LISP**

**13**. Explain how C#’s switch statement is safer than that of C.

The C# **switch** statement differs from that of its C-based predecessors in two ways.

* First, **C#** has a static semantics rule that disallows the implicit execution of more than one segment.
* The other way C#’s switch differs from that of its predecessors is that the control expression and the case statements can be strings in C#.

**14**. Under what circumstances can a linear search on a table of cases and labels be accepted?

**when the number of cases is < 10**

**15**. What mechanism does a compiler follow when the number of cases in a selection statement is 10 or greater to optimize the time required to execute?

**hash table**

**16**. What is a *loop* *variable*? What is *stepsize*?

A counting iterative control statement has a variable, called the **loop variable**, in which the count value is maintained. It also includes some means of specifying the **initial** and **terminal** values of the loop variable, and the difference between sequential loop variable values, often called the **stepsize**.

**17**. What is the difference between a pretest version and a posttest version of a logical loop?

In the **pretest** **version of a logical loop** (***while***), the statement or statement segment is executed as long as the expression evaluates to true. In the **posttest version** (***do***), the loop body is executed until the expression evaluates to false. In both cases, the statement can be compound.

**18**. In C, what is the significance of a for loop without the second expression?

**An absent second expression** is considered true, so a ***for*** without one is potentially an infinite loop.

**19**. What does the range function in Python do?

The **range() function** returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

**20**. What alternative is provided for goto in Java?

**Java does not have a goto, the loop bodies cannot be entered anywhere except at their beginnings.**

The ***break*** and ***continue*** statements allow you to jump out of a block in a loop or switch statement.

**21**. What are the design issues for logically controlled loop statements?

Because they are much simpler than counter-controlled loops, **logically controlled loops have fewer design issues.**

* Should the control be pretest or posttest?
* Should the logically controlled loop be a special form of a counting loop or a separate statement?

**22**. What is the main reason user-located loop control statements were invented?

In some situations, it is convenient for a programmer to choose a location for loop control other than the top or bottom of the loop body. As a result, some languages provide this capability.

**23**. What predefined iterator for standard data structures is available in Perl? ***foreach*** **statement**

**24**. What is the use of a **break** **statement** in switch statements?

The **break** **statement**, which is actually a restricted goto, is normally used for exiting switch statements.

**break** transfers control to the first statement after the compound statement in which it appear

**25**. What are the differences between the break statement of C++ and that of Java?

C++ has unconditional unlabeled exit with name break, while Java has unconditional labeled exit with the same name. C++ can only break the loop in which the break scope it was in, while Java can break straight to any targeted loop.

**26**. What are fatbars in guarded clauses?

The small blocks, called **fatbars**, are used to separate the guarded clauses and allow the clauses to be statement sequences.

**27**. What is the limitation of a guarded command?

that there is considerably increased complexity in the implementation of the guarded commands over their conventional deterministic counterparts.

**28**. How does a functional language implement repetition?

**Recursion method**

29.How are iterators implemented in Ruby?

**Ruby** includes iterators that are a special form of methods that are sent to various objects. The language predefines iterators for common uses, but also allows user-defined iterators.

**30**. What language predefines iterators that can be explicitly called to iterate over its predefined data structures?

**Maybe PHP**

31.What common programming language borrows part of its design from Dijkstra’s guarded commands?

**CSP**, **Haskell**

**Chapter 9 - Subprograms**

Review Questions

1.What are the three general characteristics of subprograms?

All **subprograms** **have** the following **characteristics**:

* Each subprogram has a single entry point.
* The calling program unit is suspended during the execution of the called subprogram, which implies that there is only one subprogram in execution at any given time.
* Control always returns to the caller when the subprogram execution terminates.

2.What is a subprogram call?

A **subprogram call** is the explicit request that a specific subprogram be executed.

3.What is a subprogram definition?

A **subprogram** **definition** describes the interface to and the actions of the subprogram abstraction.

4.What characteristic of Python subprograms sets them apart from those of other languages?

One characteristic of Python functions that sets them apart from the functions of other common programming languages is that function ***def*** statements are executable.

**5**. How do Ruby methods differ from the subprograms of other programming languages?

Ruby methods differ from the subprograms of other programming languages in several interesting ways.

Ruby methods are often defined in class definitions but can also be defined outside class definitions, in which case they are considered methods of the root object, *Object*. Such methods can be called without an **object receiver**, as if they were functions in C or C++.

If a Ruby method is called without a receiver, ***self*** is assumed. If there is no method by that name in the class, enclosing classes are searched, up to *Object*, if necessary.

**6**. What is the feature of Lua functions?

**All Lua functions are anonymous**, although they can be defined using syntax that makes it appear as though they have names.

**7**. What are function declarations called in C and C++? Where are the declarations often placed?

**Prototypes; in header files**

8.Name one pure functional programming language that does not have mutable data.

**Haskell**

9.What are positional parameters?

In most programming languages, the correspondence between actual and formal parameters—or the binding of actual parameters to formal parameters— is done by position:

The first actual parameter is bound to the first formal parameter and so forth. Such parameters are called **positional parameters**.

1. Give an example of a language that allows positional parameters in addition to keyword parameters.

**Python**

1. What is the use of a default value in a formal parameter?

In Python, Ruby, C++, and PHP, **formal parameters can have default values**. A default value is used if no actual parameter is passed to the formal parameter in the subprogram header

1. What is the rule of using a default parameter in C++?

In C++, which does not support keyword parameters, the rules for default parameters are necessarily different.

**The default parameters must appear last**, because parameters are positionally associated. Once a default parameter is omitted in a call, all remaining formal parameters must have default values.

1. What is the rule for accepting variable parameters in C# methods?

C# allows methods to accept a variable number of parameters, **as long as they are of the same type**.

1. What language allows array formal parameters?

**Ruby**

1. What is an ellipsis?

**ellipsis** can be treated as an array or as a list of values that can be assigned to a list of variables.

1. What are the modes, the conceptual models of transfer, the advantages, and the disadvantages of *pass-by-value*, *pass-by-result*, *pass-by-value-result*, and *pass-by-reference* parameter-passing methods?

When a parameter is **passed by value**, the value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable in the subprogram, thus implementing **in-mode** semantics.

Pass-by- value is normally implemented by **copy**, because accesses often are more efficient with this approach. It could be implemented by transmitting an **access path** to the value of the actual parameter in the caller, but that would require that the value be in a write-protected cell (one that can only be read). Enforcing the write protection is not always a simple matter.

For example, suppose the subprogram to which the parameter was passed passes it in turn to another subprogram. This is another reason to use copy transfer.

**The advantage** of pass-by-value is that for scalars it is fast, in both linkage cost and access time.

**The main disadvantage** of the **pass-by-value** method if copies are used is that additional storage is required for the formal parameter, either in the called subprogram or in some area outside both the caller and the called subprogram. In addition, the actual parameter must be copied to the storage area for the corresponding formal parameter. The storage and the copy operations can be costly if the parameter is large, such as an array with many elements.

**Pass-by-result** is an implementation model for **out-mode** parameters. When a parameter is passed by result, no value is transmitted to the subprogram. The corresponding formal parameter acts as a local variable, but just before control is transferred back to the caller, its value is transmitted back to the caller’s actual parameter, which obviously must be a variable.

The **pass-by-result** method has the **advantages** and **disadvantages** of pass-by- value, plus some additional disadvantages. If values are returned by copy (as opposed to access paths), as they typically are, pass-by-result also requires the extra storage and the copy operations that are required by pass-by-value. As with pass-by-value, the difficulty of implementing pass-by-result by transmitting an access path usually results in it being implemented by copy. In this case, the problem is in ensuring that the initial value of the actual parameter is not used in the called subprogram.

One **additional problem** with the pass-by-result model is that there can be an actual **parameter** **collision**, such as the one created with the call

*sub(p1, p1)*

In *sub*, assuming the two formal parameters have different names, the two can obviously be assigned different values. Then, whichever of the two is copied to their corresponding actual parameter last becomes the value of *p1* in the caller. Thus, **the order in which the actual parameters are copied determines their value**.

Calling a procedure with two identical actual parameters can also lead to different kinds of problems when other parameter-passing methods are used.

**Another problem** that can occur with pass-by-result is that the implementor may be able to choose between two different times to evaluate the addresses of the actual parameters: at the time of the call or at the time of the return. This makes programs unportable between an implementation that chooses to evaluate the addresses for out-mode parameters at the beginning of a subprogram and one that chooses to do that evaluation at the end. An obvious way to avoid this problem is for the language designer to specify when the address to be used to return the parameter value must be computed.

**Pass-by-value-result** is an implementation model for **inout-mode** parameters in which actual values are **copied**. It is in effect a combination of pass-by-value and pass-by-result.

The value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable. In fact, pass-by-value-result formal parameters must have local storage associated with the called subprogram. At subprogram termination, the value of the formal parameter is transmitted back to the actual parameter.

Pass-by-value-result is sometimes called **pass-by-copy**, because the actual parameter is copied to the formal parameter at subprogram entry and then copied back at subprogram termination.

**Pass-by-value-result** shares with pass-by-value and pass-by-result the **disadvantages** of requiring multiple storage for parameters and time for copying values. It shares with pass-by-result the problems associated with the order in which actual parameters are assigned.

The **advantages** of pass-by-value-result are relative to pass-by-reference. All these possible aliasing situations are eliminated if pass-by- value- result is used instead of pass-by- reference.

**Pass-by-reference** is a second implementation model for **inout-mode** parameters. Rather than copying data values back and forth, however, as in pass-by-value-result, the pass-by- reference method **transmits an access path**, usually just an address, to the called subprogram. This provides the access path to the cell storing the actual parameter. Thus, the called subprogram is allowed to access the actual parameter in the calling program unit. In effect, the actual parameter is shared with the called subprogram.

The **advantage** of **pass- by- reference** is that the passing process itself is efficient, in terms of both time and space. Duplicate space is not required and no copying is required.

There are, however, several **disadvantages** to the pass-by-reference method.

1. First, access to the formal parameters will be slower than pass-by-value parameters, because of the additional level of indirect addressing that is required.
2. Second, if only one-way communication to the called subprogram is required, inadvertent and erroneous changes may be made to the actual parameter.
3. Another problem of pass-by-reference is that **aliases** can be created. This problem should be expected, because pass- by- reference makes access paths available to the called subprograms, thereby providing access to nonlocal variables.

The **problem with** these kinds of **aliasing** is the same as in other circumstances: It is harmful to readability and thus to reliability. It also makes program verification more difficult.

1. Describe the ways that aliases can occur with pass-by-reference parameters.

There are several ways pass- by- reference parameters can create **aliases**.

* First, collisions can occur between actual parameters.

Consider a C++ function that has two parameters that are to be passed by reference, as in

***void*** *fun(****int*** *&first,* ***int*** *&second)*

If the call to *fun* happens to pass the same variable twice, as in

*fun(total, total)*

then first and second in *fun* will be aliases.

* Second, collisions between array elements can also cause aliases. For example, suppose the function *fun* is called with two array elements that are specified with variable subscripts, as in

fun(list[i], list[j])

If these two parameters are passed by reference and *i* happens to be equal to *j*, then first and second are again aliases.

* Third, if two of the formal parameters of a subprogram are an element of an array and the whole array, and both are passed by reference, then a call such as

*fun1(list[i], list)*

could result in aliasing in *fun1*, because *fun1* can access all elements of *list* through the second parameter and access a single element through its first parameter.

* Still another way to get aliasing with pass- by- reference parameters is through collisions between formal parameters and nonlocal variables that are visible.

For example, consider the following C code:

***int*** *\* global;*

***void*** *main() {*

*. . .*

*sub(global);*

*. . .*

*}*

***void*** *sub(****int*** *\* param) {*

*. . .*

*}*

Inside *sub*, *param* and *global* are aliases.

**All these possible aliasing situations are eliminated if pass-by- value- result is used instead of pass-by- reference.**

1. What is the difference between the way original C and C89 deal with an actual parameter whose type is not identical to that of the corresponding formal parameter?

**In the original C**, neither the number of parameters nor their types were checked.

**In C89**, the formal parameters of functions can be defined in two ways.

1. They can be defined as in the original C; that is, the names of the parameters are listed in parentheses and the type declarations for them follow. Using this form avoids type checking.
2. The alternative to the original C definition approach is called the **prototype** **method**, in which the formal parameter types are included in the list.

in C89, the user chooses whether parameters are to be type checked.

**19**. Name some languages that support procedures.

**Fortran** and **Ada**

1. Describe the problem of passing multidimensioned arrays as parameters.

The **problem with this method of passing matrices as parameters** is that it does not allow a programmer to write a function that can accept matrices with different numbers of columns; a new function must be written for every matrix with a different number of columns.

**21**. What is the name of the parameter-passing method used in Ruby?

**pass-by-assignment**

**22**. What are the two issues that arise when subprogram names are parameters?

1. First, there is the matter of type checking the parameters of the activations of the subprogram that was passed as a parameter.

In C and C++, functions cannot be passed as parameters, but pointers to functions can. The type of a pointer to a function includes the function’s protocol. Because the protocol includes all parameter types, such parameters can be completely type checked.

1. The second complication with parameters that are subprograms appears only with languages that allow nested subprograms. The issue is what referencing environment for executing the passed subprogram should be used.

**23**. Define *shallow* and *deep binding* for referencing environments of subprograms that have been passed as parameters.

* The environment of the call statement that enacts the passed subprogram is called **shallow binding**.
* The environment of the definition of the passed subprogram is called **deep binding**.

**24**. What is a generic subprogram?

A **generic subprogram** is one whose computation can be done on data of different types in different calls.

**25**. What is ad hoc binding?

* The environment of the call statement that passed the subprogram as an actual parameter is called **ad hoc binding**.

**26**. What causes a C++ template function to be instantiated?

**C++ template functions are instantiated** implicitly either when the function is named in a call or when its address is taken with the *&* operator.

**27**. In what fundamental ways do the generic methods of Java 5.0 differ from those of C# 2005?

Java’s generic methods differ from the generic subprograms of C++ in several important ways.

1. First, **generic parameters must be classes**—they cannot be primitive types.

This requirement disallows a generic method that mimics our example in C++, in which the component types of arrays are generic and can be primitives. In Java, the components of arrays (as opposed to containers) cannot be generic.

1. Second, although Java generic methods can be instantiated any number of times, only one copy of the code is built.
2. Third, in Java, restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters. Such restrictions are called **bounds**.

**28**. If a Java 5.0 method returns a generic type, what type of object is actually returned?

The internal version of a generic method, which is called a *raw* method, operates on *Object* class objects. At the point where the generic value of a generic method is returned, the compiler inserts a cast to the proper type.

29.If a Java 5.0 generic method is called with three different generic parameters, how many versions of the method will be generated by the compiler?

although Java generic methods can be instantiated any number of times, only one copy of the code is built.

30.When does a variable have unlimited extent?

A variable whose lifetime is that of the whole program is said to have **unlimited extent**. This usually means they must be heap dynamic, rather than stack dynamic.

**31**. What is subtype polymorphism?

**Subtype polymorphism** means that a variable of type T can access any object of type T or any type derived from T.

**32**. What is a multicast delegate?

All of the methods stored in a delegate instance are called in the order in which they were placed in the instance. This is called a **multicast delegate**.

**33**. What is the main drawback of generic functions in F#?

type inferencing and the lack of type coercion

**34**. What is a coroutine?

A **coroutine** is a special subprogram that has multiple entries. It can be used to provide interleaved execution of subprograms.

A **coroutine** is a special kind of subprogram, in which caller and called coroutines are more equitable.

**35**. What are the language characteristics that make closures useful?

language characteristics that make closures useful:

**These languages are static- scoped, allow nested subprograms, and allow subprograms to be passed as parameters.**

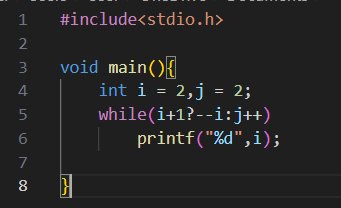
**36**. What languages allow the user to overload operators?

Operators can be overloaded by the user in **Ada**, **C++, C#** **Python**, and **Ruby**.

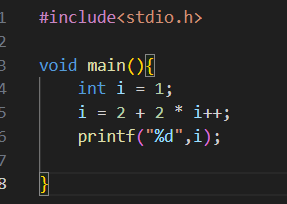
**37**. What is a symmetric unit control model? **(Page 435)**

Rather than the master- slave relationship between a caller and a called subprogram that exists with conventional subprograms, caller and called coroutines are more equitable. In fact, the coroutine control mechanism is often called the **symmetric unit control model**.

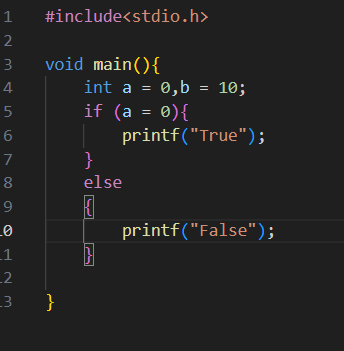
**Chapter 10 - MCQS**



output is 1



output is 4



output is False

Check the correct statement based on C Language.

select one-

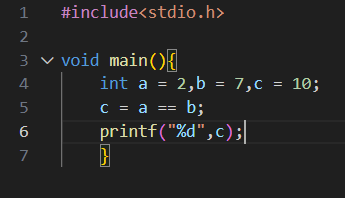
* 1. IF is a valid identifier.
  2. An identifier may start with an underscore.
  3. An identifier may end with an underscore.
  4. all of the above

Answer: b

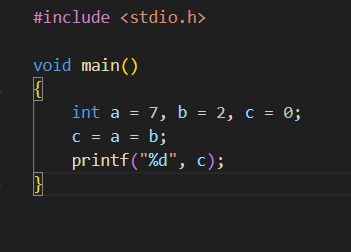
A \_\_\_\_\_\_ is a variable that is bound to a value only once.

* + - * 1. Static variable
        2. named constant
        3. literal constant
        4. implicit heap dynamic variable

Answer:b



output is 0



output is 2

In the following program a <= b will be printed if?

if (a > b)

printf(“a > b”);

else

printf(“else part”)

printf(“a <= b”)

a.a <b

b.a > b

c.doesnot depend upon if condition

d.a == b

answer:c

In the below statement ptr1 and ptr2 are uninitialized pointers to int i.e they are pointing to some random addresses that may or may not be valid addresses.

int\* ptr1, ptr2;

a.false, because they are pointing to null address.

b.true, because they are in the same statement

c.true, because they are not initialized

d.false, because they are different type variables

Answer:d

Direct descendents of C alongside with itself do not allow subprogram nesting.

a.False,except C.

b.True, except C

c.True for all

d.False for all

Answer:b

In the following program fragment, s2 will be executed if?

if (a > b)

if (b > c)

s1;

else

s2;

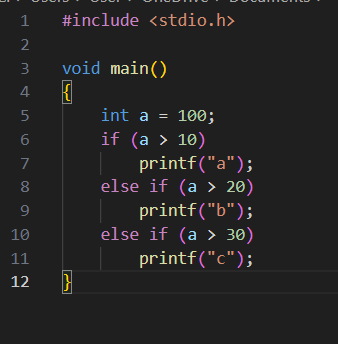
a . a > b and b <= c

b. a <= b

c.b <=c and a <=b

d. b > c

answer:b



output is a