English is very flexible: there are often many ways to say something. Programming languages have this same flexibility, as is demonstrated by the tremendous variety in the solutions handed in for one student programming problem.

Possible to translate them with regularity into machine language

Compilers can't correct more than trivial semantic errors, those that attempt to make a tradeoff of time/space.

Structural Deviations: In English, they are common. Programming langs: compilers are not very tolerant of deviations.


15. Redundancy: English = very redundant, and meaning can be ascertained despite many errors. Prog langs = some redundancy assists us in locating errors. (i.e. overloading functions, blocks, explicit/duplicate type declarations)

16. Implicit communication: English = reading between the lines. Prog langs = use of defaults.

20. Abstraction: Both English and prog langs afford us a great deal of abstraction.

21. Flexibility: Both English and programming languages are flexible: many ways to say the same thing. The various differences (syntax, different algorithms with varying time/space complexity, etc) are sometimes called nuances.

24. Standardized language: only one dialect. Disadvantage of languages with many dialects: can be non-portable.

27. The syntax specifies how individual words may be written and the order in which words are placed within a sentence. The semantics specifies the meaning attached to the logical meaning of a word in a sentence, the meaning of omitting a sentence element, and the meaning of each individual word. Programs must conform to very strict structural rules that govern the order of statements and sections of code and particular ways to begin, punctuate and end every program. A compiler will identify an error when the input text fails to correspond to the syntactic rules of the language or when an object is used in the wrong context (a "type error"). Programming languages are partly redundant, and the required redundancy serves as a way to identify errors. A programmer learning a new language must learn its implicit assumptions, more commonly called defaults. If a programmer relies on defaults to convey meaning, the compiler cannot tell the difference between the purposeful use of a default and an accidental omission of an important declaration. Stating information explicitly is less error prone and enables a compiler to give more helpful error comments. When different statement sequences in a programming language express the same algorithm, they differ in subtle ways, such as the time and amount of memory required to execute the algorithm. This is called different nuances. Programs written in one dialect must be modified to be used by people whose computer "understands" a different dialect. When this happens, we say a program is nonportable. The cost of rewriting programs makes nonstandardized programming languages unattractive to commercial users of computers; the language specifications and reference material must be reread and rewritten for each new dialect.

28. Syntax is the set of rules, principles, and processes that govern the structure of sentences in a given language. Semantics is the study of meaning that is used for understanding human expression through language. The differences between English and programming languages are real, but not as great as they might at first seem. The differences are less extreme now than they were ten years ago and will decrease as programming languages continue to evolve.

31. The syntactic structure of English is highly redundant. The same information is often conveyed by several words or word endings in a sentence.

32. Redundancy repetition of messages to reduce the probability of errors in transmission.
3.2 Metalanguage

An assortment of syntax delimiters, metawords, and definitions and ways to refer to structural units.

3.2.4 to 4.4

Lambda Calculus—formula written in symbols manipulated according to logical rules

4.4 Lambda Calculus

Formulas: single variable, F(defined), $\lambda y.F$ (lambda expression), FG (application)

Lambda Calculus defines a minimal semantic basis for computation. Using variables, lambda expressions, and applications, lambda calculus can represent program computations.

Chapter 4

4.3.4 to 4.4

Lambda Calculus—formula written in symbols manipulated according to logical rules

4.4 Lambda Calculus

Formulas: single variable, F(defined), $\lambda y.F$ (lambda expression), FG (application)

Lambda Calculus defines a minimal semantic basis for computation. Using variables, lambda expressions, and applications, lambda calculus can represent program computations.

Chapter 3 - Elements of Language

3.1 Nouns

In natural languages nouns give us the ability to refer to objects

3.2 (up to comments):

Metalanguage permits a language to talk about itself. ex "the paragraph below"

Lexic Tokens are names, nums, strings, & symbols. They usually separated by spaces.

but not always separated by spaces. ex "bar=foo*35;" has 6 tokens and no spaces.

The lexer is the first step of the compiler. It breaks text into Lexic Tokens

98: Putting a symbol (commonly ";") denotes the end of a sentence

99: Scopes (paragraphs) can be defined with brackets and/or statements (ie. if, else, or)

100: 3.2 Pt 2

101: Old languages used specific markers in a fixed position, which limited usefulness

102: Comment "overflow" (with block comments) cannot be detected by translator

103: Complete language has single-line, partial-line (//) and block (/**/)

104: Filenames, line numbers exemplify metawords that describe parts of a program

105: Symbol Table or Dictionary - the set of syntactic words the compiler recognizes

106: Preprocessor adds syntax without semantics—you cannot add syntax functionality

107: Lexical analysis must be done before macro expansion to identify macros, as well as after expansion during compilation.

108: C did not specify if tokenization is done before/after preprocessing. Problem

109: because quoted strings must be one token so that they are not edited inside.

110: ANSI C specified tokenization is done first to solve this problem.

111: ANS C3 specified tokenization is done first to solve this problem.

112: ***

113: 3.1 Parts of Speech

114: 3.1.1

115: Nouns

116: Variable declaration:

117: - Setting aside storage to represent a real world object

118: - First Class Objects - Objects that can be manipulated and processed as whole units.

119: - giving the memory a name so that the it can be accessed.

120: In different programming languages a name can:

121: Exist without being bound to an object

122: Be bound to several objects in different scopes simultaneously

123: Be bound to different objects at different times

124: Be bound through a pointer to an object that doesn’t exist

125: 3.1.3 Adjective Data Types

126: Corresponds to attributes that are associated with an object

127: 3.1.4 Verbs

128: Includes: Functions, Procedures, and Operators

129: Comments

130: Can be added to convey general information about the text or program.

131: 3.2 Metalanguage

132: Assortment of syntax delimiters, metawords, and definitions and ways to refer to structural units.

133: (Lexical tokens, Statements, Scope, Comments)

134: **

135: **** Chapter 4

136: 4.3.4 to 4.4

137: Lambda Calculus—formula written in symbols manipulated according to logical rules

138: 2 symbol types: single character (variable) and punctuation (' ',' ',' and '?')

139: Formulas: single variable, F(defined), $\lambda y.F$ (lambda expression), FG (application)

140: Variable directly after ?? is bound, any occurrence after is bound to ??, else free

141: eta reduction: $\lambda x.f(x) \rightarrow f$ where $f(x)$ is a function

142: Reducing: renaming and substituting until it is put into normal form.

143: Extension: set of definitions which augment a language with a new facility that can be used in the same way that preexisting facilities are used (eg new types)

144: Early FORTRAN had only 2 data types, and could not be extended

145: C++, an extension of C by changing the compiler to add virtual functions and more

146: In C++ syntactic extension is done without changing the compiler

147: EL/1 FORTH and T allow changing the compiler for syntactic extension

148: ***

The rules for constructing well-formed statements, called the language’s syntax, are commonly displayed in Extended Backus-Naur Form (EBNF) or syntax diagrams. An EBNF grammar has a starting symbol, a set of terminal symbols, a set of nonterminal symbols, and a series of rules.

- One or the other: $s ::= A | B$ is 's may be replaced by $B$ or $AB$ or $AAB$ etc.

- Zero or more times: $s ::= \{A\}B$ is 's may be replaced by $B$ or $AB$ or $AAB$ etc.

- While syntax describes which combinations of symbols lead to a legal program, semantics describe the actual implementation of those symbols. The semantic basis of a language can be described by a specific version of the abstract machine (program environment, stack, streams, shared environment, control).

150: Lambda calculus defines a minimal semantic basis for computation. Using variables, lambda expressions, and applications, lambda calculus can represent program computations.

151: - One or the other: $s ::= A | B$ is 's may be replaced by $B$ or $AB$ or $AAB$ etc.

152: - Zero or more times: $s ::= \{A\}B$ is 's may be replaced by $B$ or $AB$ or $AAB$ etc.

153: To generate a program using a grammar, expand the starting symbol according to the production rules. After all nonterminals have been expanded, the result is a grammatically correct program. Syntactic analysis, or parsing, translates source code into a parse tree representing the structure of the code. If the code cannot be built into a parse tree, it is not grammatically correct. A syntax diagram will show the path that each statement takes through the production rules.

154: While syntax describes which combinations of symbols lead to a legal program, semantics describes the actual implementation of those symbols. The semantic basis of a language can be described by a specific version of the abstract machine (program environment, stack, streams, shared environment, control).

155: Lambda calculus defines a minimal semantic basis for computation. Using variables, lambda expressions, and applications, lambda calculus can represent program computations.


157: - Rules: 'y' (variable), $\lambda y.F$ (expression), and 'FG' (application) are formulas

158: A language's semantics are extendable if they may be augmented with new actions, data types, or control structures.

159: ***

160: Syntax

161: The syntax of a language is its grammatical rules.

162: These are usually defined through EBNF (Extended Backus-Naur Form) or syntax diagrams.

163: The meaning of a program is represented by a program code or by a computation tree.

164: The language syntax defines the computation tree that corresponds to each legal source program.

165: The rules for constructing a well-formed sentence (statement) out of words, a paragraph(module) out of sentences, and an essay (program) out of paragraphs are the syntax of the language.

166: It is usual to define the syntax of a programming language in a formal language. Two most common formalisms are Extended Backus-Naur Form (EBNF) and syntax diagrams.

167: EBNF

168: 1. An EBNF language definition can be translated by a program called a parse
Lambda Calculus

Two kinds of symbols:

- A single-character symbol, such as 'y', used to name a parameter and is called a variable.

- Punctuation symbols '(', ')', ',', etc. These symbols can be combined into strings to form formulas according to these simple rules:

  - 1. A variable is a formula.
  - 2. If y is a variable and F is a formula, then ('lambda_symbol':y:F) is a formula, which is called a lambda expression 'lambda_symbol':y:F is said to be the parameter of the lambda expression, and F is its body.
  - 3. Its output is the list of the symbols defined in the program and a parse tree, which specifies the role that each source symbol is serving.

Two kinds of symbols: (part of speech) of every source symbol and combination of symbols.

A single-character symbol, such as 'y', used to name a parameter and is called a variable.

Punctuation symbols '(', ')', ',', etc. These symbols can be combined into strings to form formulas according to these simple rules:

1. A variable is a formula.
2. If y is a variable and F is a formula, then ('lambda_symbol':y:F) is a formula, which is called a lambda expression 'lambda_symbol':y:F is said to be the parameter of the lambda expression, and F is its body.

Thus every lambda calculus formula is of one of three types: a variable, a lambda expression, or an application.

Free and Bound Variables:

- A parameter name is a purely local name. It binds all occurrences of that name on the right side of the lambda expression.

- A symbol on the right side of the lambda expression is bound if it occurs as a parameter, immediately following the lambda symbol, on the left side of the same expression or of an enclosing expression. Each bound occurrence of x refers to the particular lambda:x that binds it.

Chapter 5 -- Primitive Types

A Data type is an abstraction. All data types must be mapped onto bytes/words of the machine.

Specific (concrete) types = homogeneous set of objects, Generic types = objects w/ 2+ types.

Every language supports a set of primitive data types, differs dependent on intended hardware.

- Primitive types originally built into instruction sets, Ada language treated types like objects.
- C: 5.1
- Java: 5.2
- C++: 5.3
- C#: 5.4
- C: 5.5
- Python: 5.6
- Smalltalk: 5.7
- Eiffel: 5.8
- Ada: 5.9
- C++: 5.10
- Java: 5.11

Type is an abstraction -- it is a common property of a set of similar data objects.

Type Declaration -- defines type name and associates type description with id' or a 'bcd'.

Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

-type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstraction -- it is a common property of a set of similar data objects. Primitive Types -- defined by system implementor. Otherwise programmer defines.

- Type is an abstract...
Chapter 7 -- Names and Binding:

The meaning of a name is its binding and a name must be given a meaning before it can be used.

Names cannot be created dynamically in most languages.

The symbol table is a data structure that stores names and their definitions during translation.

Primitive symbols are the names of the built-in functions, types, and constants and their definitions.

These names do not occur in the language's syntax.

In typed languages, each name defined in a program unit has a fixed data type associated with it.

FORTH permits the user to redefine a word that is already in the symbol table (dictionary).

With fully dynamic binding, the type must be stored with the object in memory ("typeless").

Constants may be implemented by pre-calculating its value before compilation and substituting that value in the source code, or by creating an initialized read-only variable (IROV).

Lexical scoping: the complete name to which a use refers is the one produced by the nearest declaration for it.

The nearest declaration is the one in the smallest enclosing block.

Redeclaration of an identifier locally will mask any definitions of the same identifier that are relatively global to it.

Built-in functions and comparison work with primitive types but user defined types are not as convenient or easy to use.

Some languages implement non-hierarchical sharing to break the block-structured paradigm.

The decision to make a type a primitive type: cost and added feature complications.

If typical hardware does not provide instruction to handle the type it may be costly and inefficient to implement.

User implementation execute less efficiently than system implementation.

Language structure may be unable to support the type with user implementat...
346: Call-by-value: the simplest and cleanest parameter passing mechanism, both in terms of its semantics and its implementation.

348: Call-by-Reference:The call-by-reference parameter passing mechanism is also called call-by-address and, in Pascal, VAR parameter passing.

349: Call-by-Return: 1. So that a value may be returned. 2. To avoid consuming the execution time and stack space necessary to pass a very large argument by value.

351: Call-by-pointer: a subcase of call-by-value. The contents of the pointer variable in the calling program is an address.

352: A higher-order function: one that takes a function as an argument or returns a function as its argument.

353: result.

354: Mapping Functions. Frequently, we have a monadic or dyadic operation that we wish to apply to a list or array of data objects.

355: to a list or array of data objects.

356: Currying:

357: In functional languages, every function of two or more arguments is considered to be a higher-order function of one argument that returns another function.

358: ***

359: 9.1 Function Syntax: Many languages’ functions require a fixed number of parameters when declared, but some don’t. One option is optional parameters, where some parameters can be left out. This has the danger though of making the code hard to understand but may reduce a lot of unnecessary values that are always the same. The other option is indeterminate parameter passing, where an argument is passed by a default value, where it is used. This way of passing parameters can be static; the name is bound to the object when the object is allocated and remains bound throughout the program.

360: When there isn’t a one-to-one correspondence between a name and object -- it will be recognized by the translator. Scope can be global, in which case the name is known throughout the program, or it can be local, meaning that it is only known within that program block in which it was defined.

361: In a block structured language, a name becomes visible when it is born and invisible again when it dies. However, a living name also becomes temporarily invisible when it is passed to, for example, a subroutine.

362: Compiled Language Systems: enable the programmer to enter, compile, and link/load programs; when execution is done, control returns to the OS or to the shell.

363: Currying: In functional languages, every function of two or more arguments is considered to be a higher-order function of one argument that returns another function.

364: Call-by-name: make the parameter values stand for the parameters given. Dangerous as the values inside function can be changed unexpectedly.

365: Call-by-reference: make pointer to values that are passed to, this process changes the values but not the pointers made.

366: Call-by-return: able only to change parameter value, not read it.


368: Call-by-pointer: same as reference but you pass pointer address, this has danger of pointers pointing to null.

369: Chapter 9: Implications: Closures are a useful device for taking general, i library code modules and tailoring them to the needs of a particular environment.

Chapter 9: Indefinite-length argument lists: Parameters might be passed but not named.

374: Closures: Closure is an operation that can be applied to an expression. We lose an expression by binding each of its free variables to the value of another expression. A closure creates a function, which is returned as the result of the closure.

375: Implementations: For taking general, i library code modules and tailoring them to the needs of a particular environment.
376: ***
377: **** Chapter 10:
378: Chapter 10.1
379: Basic Control Structures: A control structure is a language feature defined in the semantics of the language (not in syntax only) that defines the order of evaluation of expressions or execution of program units. Execution starts when the computer is turned on or receives a RESET signal.
380: Normal Instruction Sequencing: Normally, instructions are executed in the order that they are loaded into memory. This is carried out by the instruction cycle of the machine. Machine instructions are executed in the order in which they are loaded into memory unless that order is changed by a jump instruction.
381: Assembler: Two kinds of control structures above the expression level are basic: subroutine call with parameters and conditional execution (discussed in Section 10.2). Procedural languages have two more basic kinds of control: execution of a sequence of statements and repetition.
382: Sequence, Subroutine Call, IF, and WHILE Sufficient: Each of these control structures is comprised of a group of boxes with connecting arrows. Each group is a one-in-one-out structure; that is, on a flowchart, one line comes into the control structure and one line goes out of it. This is an important property: programs limited to these formation patterns are easier to debug because the effects of any transfer of control are confined to a small and defined part of the algorithm.
383: Call: This instruction, also called jump to subroutine, saves the current value of the IC, then stores the entry address of the subroutine in the IC. Subroutine return restores the saved address and brings control back to the instruction after the call. This methodology, of decomposing a problem into clearly defined parts with limited and clearly defined interactions, then building a program by implementing each, is called "top-down programming". It is now recognized to be the basis of good program design.
384: Jump and Conditional Jump: Jump and conditional jump instructions change the next instruction to be executed by storing a new address into the IC. They differ from a jump to subroutine instruction in that they do not save or restore the address from which the jump originated. Control passes to (b) after execution.
385: Control Diagrams: In order to visualize and compare the wide range of control statements in different languages, we need a language-independent representation for each basic kind of control. Flowcharts provide a graphic representation of control flow, but cannot represent the difference between GOTO and structured transfers of control.
386: 10.2 Conditional Control Structures
387: Conditional expressions return some result, such as b>c?b:c; however, condition statement renge requires no results; also, they can be transformed to each other. We can improve the lexical coherence of IF-ELSE by emulating a structured IF THEN ELSE. If or structured conditions, we can use THEN...ELSE, which makes programs have fewer keywords and language syntax simpler but causes ambiguity in the dangling else situation. The IF statement has the form IF (condition) THEN (actions) ELSE (actions) ENDIF, which makes programs not fit with BEGIN ...END pairs and solves the ambiguity problem. Else statement can be replaced by nested if. Case statement evaluates an expression and uses the result to select the label, which is really efficient. C switch differs from the general CASE because a program will continue to next label without breaking.
388: 10.3 Iteration
389: Infinite loop is used to implement processes that interact by using the IO interrupt system. The WHILE loop in general combines WHILE loop and REPEAT loop and supports Until test and While test. For a counted loop, the initial values of the loop variable, the goal value and the increment value should be fixed before execution, otherwise the number of iteration is unpredictable. The most general form of the loop is WHILE loop which is called the iteration element, and FOR loop in C is a counted loop. When using IF condition, there is no end condition of a loop. The loop control element is not necessarily after the loop, an expression whose result must be interpreted as a truth value and an increment after the scope of the loop.
390: 10.4 Implicit Iteration
391: Implicit looping is powerful to reduce the line of coding and flexible. Backtracking is used in some pattern matching languages, such as Prolog. The implementation of backtracking uses a stack to store pointers to the positions at which each match trial was made.
392: ***
393: **Iteration**
394: Iteration is explicit in procedural languages and implicit in functional languages.
395: Structured Loop: (one-in, one-out)
448: Related sections of code are frequently not grouped together, due to moving a GOTO target label to somewhere else. Programmers often forget to complete far-off sections, and debugging is more tedious.

449: While GOTO's slow down translation and make code messier, they are also early times the only efficient or practical way to implement a control pattern. The GOTO's target label can be implemented using line numbers or defined labels.

450: Many looping situations require both finishing naturally and aborting early. The BREAK statement is simply a GOTO statement targeting the next line of code out side of the block. To escape from nested loops, FAR_EXIT can be implemented to GOTO the outside of the longest loop.

451: A continuation is a "checkpoint" containing everything that is needed in order to safely restart execution at a later time (program counter, stack, environment vars, etc.)

452: When exceptions occur, which include hardware errors, software errors, and logical inconsistencies, robust systems should identify and control the effects as much as possible. Hardware errors can be detected through system calls and error codes, error propagation passes exception information to the exception handler, and the handler takes action before resuming the continuation.

453: ***

454: Continuations

455: - Functional languages do not have sequences of statements, but need a way to implement same functionality as in procedural languages.

456: - Continuation: a function which "acts like" the remainder of the program to still be executed.

457: - Concept of continuation exists in all programming languages, but higher-order functional languages give programmers explicit access.

458: - Packaging a continuation is like establishing a checkpoint; includes program counter, stack, environment vars, etc.

459: Exceptions

460: - How exceptions arise: 1) Hardware error trigger interrupt signal, 2) System identifies an error (e.g. subscript outside of bounds), 3) User function identifies an error situation by raising an error signal.

461: - Ignoring exceptions discourages locality of effects.

462: - Hardware detects exception and generates an interrupt signal which is processed by OS. OS will then set status flags.

463: - Software Exception: some languages provide a general exception handling construct structure.

464: - Passing control: When exception occurs it is more useful to propagate error to handler by passing by value. "spaghetti code" is that virtually everything has global effects.

465: - Downside of above is error handling code is interleaved with normal code, and intermediate routines need to have propagation code even though they have nothing to do with the error.

466: - Propagate by popping stack frames until handler is found or it returns to system.

467: - Handler code is translated in the context of its enclosing block.

468: - Being able to raise an exception by a specific name, provides more context about the cause of the problem.

469: ***

470: When the GOTO was first introduced into higher-language languages, it was thought to be both basic and necessary. It was considered basic because it directly reflected the branch instructions of all computers. It was considered necessary because it was used to compensate for the nonexistence of an important semantic mechanism.

471: We can divide the faults of the GOTO control structure roughly into three categories: bad notation, bad effects on proofs of correctness, and bad human engineering properties.

472: Goto is not a problem. The problem is many people who use it.

473: Many programmers consider using the GOTO construct bad practice because they think it leads to spaghetti code. Spaghetti code is that virtually everything has global effects. A spaghetti program has a short useful lifetime and poor portability.

474: Many of the newer computer languages contain no GOTO at all. The omission of GOTO is particularly interesting in the cases where a new language is basically a revision of an old one, as Icon is a revision of SNOBOL.

475: Most languages avoid these problems by having a programmer simply write a label at the beginning of any line. In Pascal, the label must also be declared at the top of the program. The uses and misuses of GOTO and statement labels have been considered by a variety of ways to minimize the use of GOTO.

476: Many loops have two possible reasons for termination, not one. The basic search loop is a good example: control will leave the loop after all data items have been searched and the key item is not found. The BREAK statement takes control immediately out of the enclosing control structure, and into the control scope that surrounds it.

477: There are three major ways in which exceptions arise:

478: 1- A hardware error occurs and triggers a hardware interrupt signal.

479: 2-A system software module identifies an error situation.

480: 3-A user function identifies a logically impossible or inconsistent situation.
508: Prolog useful when programmer does not know how to organize data or computational process