Chapter 1 - The Nature of Language:

Semantics: meaning
Semantic intent: the idea an agent wishes to communicate
Programming language: communication between computers and humans. It must be possible to translate them with regularity into machine language
Structured programming: high level languages are more flexible
Compilers can't correct more than trivial semantic errors, those that attempt make a tradeoff of time/space.
Restructural Deviations: in English, they are common. Programming languages: compilers not translate structures. Syntax errors are detected, bad restrictions prevent nonsense; bad limits the programmer; powerful languages can express many actions

Principle of Frequency: more frequent functions should be easier to use
Principle of Locality: good languages enforce locality (closeness) of effects
Global variables (vars), locals, encouraged, no globals
Principle of Lexical Coherence: logical blocks of code should be next to eachother
Principle of Distinct Representation: each object should have its own item
Principle of Portability: able to be compiled on many different platforms
Principle of Modularity: small programs are implemented as functions; strongly typed: objects have names and types; object-oriented: have classes; procedural languages: sequential code; new functional languages: variables dont exist
Parallel: support for threading

Latex: support for rendering

**** Chapter 3 - Elements of Language

3.1: Nouns:
In natural languages nouns give us the ability to refer to objects
First-Class Objects: an executable piece of code, has begun to achieve first-class status in some languages, which are known as "functional languages"
Naming Objects: A variable is a name for an object (LISP).

Pronouns: The most important use of pronouns in programming languages is to label objects that are dynamically created
Adjectives: Type data which attributes can be associated with an object by a declaration
Verbs: Procedure calls, function calls, and arithmetic operators all direct that some action should happen, and are like action verbs
Prepositions and Conjunctions: Each programming language contains a small number of such words, but not usually to delimit phrases and denote choice and repetition (WHILE, ELSE, BY, CASE, etc.)

3.2 (up to):
Lexicon: a language to talk about itself, ex the paragraph below
Lexicon Tokens are names, nums, strings, symbols. They usually represent 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 Lexicon Tokens
Putting a symbol (commonly *) denotes the end of a sentence
Scopes (paragraphs) can be defined with brackets and/or statements (ie. if, for)
Old 2 Pt 2
Old languages used specific marks in a fixed position, which limited usefulness
Compositions of the alphabet into building blocks (tokens)
Complete language has single-line, partial-line // and block /***/
Filenames, line numbers exemplify metawords that describe parts of a program
Symbol Table or Dictionary- the set of syntactic words the compiler recognizes
Preprocessor adds syntax without semantics-you cannot add language functionality
Lexical analysis must be done before macro expansion to identify macros, as well as after expansion during compilation.
C did not specify if tokenization is done before/after preprocessing. Problem because quoted strings must be one token so that they are not edited inside.
ANSI C specified tokenization is done first to solve this problem.

3.1 Parts of Speech

3.1.1 Nouns

Variable declaration:
- Setting aside storage to represent a real world object
- First class Objects - Objects that can be manipulated and processed as whole units.
- Giving the memory a name so that it can be accessed.

In different programming languages a name can:
- Not exist without being an object
- Be bound to several objects in different scopes simultaneously
- Be bound to different objects at different times
- Be bound through a pointer to an object that doesn't exist

3.1.3 Adjective Data Types

Corresponds to attributes that are associated with an object

3.1.4 Vars

Includes: Functions, Procedures, and Operators

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

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

(Lexical tokens, Statements, Scope, Comments)

Chapter 4
4.3.4 to 4.4

Lambdas- formulas written in symbols manipulated according to logical rules.

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

Formulas:
- single variable, F(defined), ??F (lambda expression), P(F) (application)
- Variable directly after ?? is bound, any occurrence after is bound to ??, else free eta reduction: ??x.f(x) -> f where f(x) is a function.

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

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 type)

Early FORTRAN had only 4 data types, and could not extend C++

An extension of C by changing the compiler to add virtual functions and more in C++ syntactic extension is done without changing the compiler

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

The rules for constructing well-formed statements, called the language’s syntax, are
commonly display 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. Notation:
- One or the other: s ::= A | B is ‘s may be replaced by B’
- Zero or more times: s ::= (A)B is ‘s may be replaced by B or AB or AAB etc.’

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.

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

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

Symbols:
- single-character ‘y’ (variable), or punctuation ‘(‘, ‘)’, ‘{’, ‘}’, ‘?’
- Rules: ‘y’ (variable), ‘??y.F’ (expression), and ‘P(F)’ (application) are formulas
- A lambda expression is extendable if they may be augmented with new actions, data types, or control structures.
even details that were never considered by the language designer or semantics writer. Precision and completeness are more important for this purpose than readability, and formal semantic definitions are not easy to read. A language may be extended through its vocabulary and its syntax. Lambda Calculus

Two kinds of symbols:

1. A single-character symbol, such as ‘$’, used to name a parameter and is called a variable.
2. A group of such symbols is called a lambda expression. For instance, ‘$x + y’ is a lambda expression, $x$ and $y$ are the parameters of the lambda expression.

These symbols can be combined into strings to form formulas according to three simple rules:

1. A variable is a formula.
2. If $y$ is a variable and $F$ is a formula, then ‘($y$’ symbol $y$:F) is a formula, which is called a lambda expression.
3. If $F$ and $G$ are formulas, then ‘($F$’ symbol $G$) is a formula, which is called an application.

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 an enclosing expression. Each bound occurrence of $x$ refers to the particular lambda $x$ that binds it.

**Chapter 5 -- Primitive Types**

5.0 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 not.

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

Primitive types originally built into instruction sets, Ada language treated types like objects.

5.1 Computer memory is long array of bits, organized into groups each w/ an address. Logical operations occur on uninterpreted bytes, e.g. left/right shifts.

Character codes: Hollerith was first char set, then EBCDIC, then ASCII later formed.

Numbers: Packed Decimals, (Signed/Unsigned) Binary Integers, Floating Point. Binary Integers: String of digits, each decimal digit rep. by 4 bits, pairs of digits in bytes. Binary Integers a primitive type of C, Floating Point adheres to IEEE Standard. IEEE ensures maximum accuracy but its range of numbers can be inconsistent e.g. $0.1 + 0.1 = 0.10000000000000001$.

Type is an abstraction -- it is a common property of a set of similar data objects. Primitive types -- defined by system implementor. Otherwise programmer defined. Type Declaration -- defines type name and associates type description with it. If all objects in a type have same size, structure, & semantic intent, then they are concrete.

Generic domain is a set that includes objects of more than one concrete type. A specific type that is included in a generic domain is called an instance of the generic domain. E.g. Specific Types = Int arrays, Int numbers. Generic domains = Set of Int arrays of all lengths.

Omit Primitive Type Cost: Inefficiency, cannot extend built-in generic funcs, lang not extensible.

5.3 Types that cannot be supported by hardware could be emulated using software. Business PC memory supports records/tables/arrays, scientific PC mem only supports arrays.

**Chapter 6 -- Modeling Objects:**

Assignment is an operation that stores a program object into an existing storage object. Called destructive assignment because the previous contents of the storage object are lost.

Implicit assignment: when you can assign compound values to a compound object (such as an array)

- Multiple assignment: when you can write a single expression which assigns a value to several storage objects.
- Dereferencing by system implementor. Otherwise programmer defined.
- Type Declaration -- defines type name and associates type description with it.
- If all objects in a type have same size, structure, & semantic intent, then they are concrete.
- Generic domain is a set that includes objects of more than one concrete type. A specific type that is included in a generic domain is called an instance of the generic domain. E.g. Specific Types = Int arrays, Int numbers. Generic domains = Set of Int arrays of all lengths.
- Omit Primitive Type Cost: Inefficiency, cannot extend built-in generic funcs, lang not extensible.

5.3 Types that cannot be supported by hardware could be emulated using software. Business PC memory supports records/tables/arrays, scientific PC mem only supports arrays.

**Chapter 6 -- Modeling Objects:**

Assignment is an operation that stores a program object into an existing storage object. Called destructive assignment because the previous contents of the storage object are lost.

Implicit assignment: when you can assign compound values to a compound object (such as an array)

- Multiple assignment: when you can write a single expression which assigns a value to several storage objects.
- Dereferencing by system implementor. Otherwise programmer defined.
- Type Declaration -- defines type name and associates type description with it.
- If all objects in a type have same size, structure, & semantic intent, then they are concrete.
- Generic domain is a set that includes objects of more than one concrete type. A specific type that is included in a generic domain is called an instance of the generic domain. E.g. Specific Types = Int arrays, Int numbers. Generic domains = Set of Int arrays of all lengths.
- Omit Primitive Type Cost: Inefficiency, cannot extend built-in generic funcs, lang not extensible.

5.3 Types that cannot be supported by hardware could be emulated using software. Business PC memory supports records/tables/arrays, scientific PC mem only supports arrays.

**Computer Memory:** Computer memory are arrays of bits organized into group. These bits are grouped into 8 to form a byte. 2 and 4 bytes are called a word and a long word.

Theses bytes and words form the basis for all computation.

Character Code: Common encoding such as ASCII, EBCDIC are imposed on bit strings to impose meaning beyond the bits. EX: ASCII uses 7 bits to represent 128 characters.

Packed Decimal: (Varies between machine) It’s used to implement decimal fixed-point arithmetic which has two integer fields. One represents the magnitude the other the position of the decimal point.

Negative Numbers: There are several ways to represent a negative integer. For example, one bit can be interpreted as the sign and the rest as the magnitude or with tw o’s complement.

Floating Point: The IEEE has a standard for floating-point representation and computation which supports float of three lengths: 4, 8, and 10 bytes. The standard covers all aspects of floating point. The sign bit is always at the left end. Floating points also have an exponent and mantissa part.

Data Type: Abstraction of common expression for a set of similar objects and these properties help define a representation for these objects.

Type name: a name associated with a type description.

Typed declaration: is used to define a name and a type associated with it.

Specific type: a homogenous set of objects.

Generic domain: is a set of objects that includes more than one concrete type.

Specific type example (Real numbers, Integers)

Generic domains example (numbers: This includes floating point, integer, and packed decimal).

Decision to make a type a primitive type: Cost

Added feature complicate the language syntax and semantics.

The compiler/library/runtime system become more complex.

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

Cost of omitting type:

User implementation execute less efficiently than system implementation.

Language structure may be unable to support the type with user implementation.

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

**Chapter 6 -- Modeling Objects:**

Assignment is an operation that stores a program object into an existing storage object. Called destructive assignment because the previous contents of the storage object are lost.

Implicit assignment: when you can assign compound values to a compound object (such as an array)

- Multiple assignment: when you can write a single expression which assigns a value to several storage objects.
- Dereferencing by system implementor. Otherwise programmer defined.
- Type Declaration -- defines type name and associates type description with it.
- If all objects in a type have same size, structure, & semantic intent, then they are concrete.
- Generic domain is a set that includes objects of more than one concrete type. A specific type that is included in a generic domain is called an instance of the generic domain. E.g. Specific Types = Int arrays, Int numbers. Generic domains = Set of Int arrays of all lengths.
- Omit Primitive Type Cost: Inefficiency, cannot extend built-in generic funcs, lang not extensible.

5.3 Types that cannot be supported by hardware could be emulated using software. Business PC memory supports records/tables/arrays, scientific PC mem only supports arrays.

**Computer Memory:** Computer memory are arrays of bits organized into group. These bits are grouped into 8 to form a byte. 2 and 4 bytes are called a word and a long word.

Theses bytes and words form the basis for all computation.

Character Code: Common encoding such as ASCII, EBCDIC are imposed on bit strings to impose meaning beyond the bits. EX: ASCII uses 7 bits to represent 128 characters.

Packed Decimal: (Varies between machine) It’s used to implement decimal fixed-point arithmetic which has two integer fields. One represents the magnitude the other the position of the decimal point.

Negative Numbers: There are several ways to represent a negative integer. For example, one bit can be interpreted as the sign and the rest as the magnitude or with tw o’s complement.

Floating Point: The IEEE has a standard for floating-point representation and computation which supports float of three lengths: 4, 8, and 10 bytes. The standard covers all aspects of floating point. The sign bit is always at the left end. Floating points also have an exponent and mantissa part.

Data Type: Abstraction of common expression for a set of similar objects and these properties help define a representation for these objects.

Type name: a name associated with a type description.

Typed declaration: is used to define a name and a type associated with it.

Specific type: a homogenous set of objects.

Generic domain: is a set of objects that includes more than one concrete type.

Specific type example (Real numbers, Integers)

Generic domains example (numbers: This includes floating point, integer, and packed decimal).

Decision to make a type a primitive type: Cost

Added feature complicate the language syntax and semantics.

The compiler/library/runtime system become more complex.

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

Cost of omitting type:

User implementation execute less efficiently than system implementation.

Language structure may be unable to support the type with user implementation.

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

**Chapter 6 -- Modeling Objects:**

Assignment is an operation that stores a program object into an existing storage object. Called destructive assignment because the previous contents of the storage object are lost.

Implicit assignment: when you can assign compound values to a compound object (such as an array)

- Multiple assignment: when you can write a single expression which assigns a value to several storage objects.
- Dereferencing by system implementor. Otherwise programmer defined.
- Type Declaration -- defines type name and associates type description with it.
- If all objects in a type have same size, structure, & semantic intent, then they are concrete.
- Generic domain is a set that includes objects of more than one concrete type. A specific type that is included in a generic domain is called an instance of the generic domain. E.g. Specific Types = Int arrays, Int numbers. Generic domains = Set of Int arrays of all lengths.
- Omit Primitive Type Cost: Inefficiency, cannot extend built-in generic funcs, lang not extensible.

5.3 Types that cannot be supported by hardware could be emulated using software. Business PC memory supports records/tables/arrays, scientific PC mem only supports arrays.

**Computer Memory:** Computer memory are arrays of bits organized into group. These bits are grouped into 8 to form a byte. 2 and 4 bytes are called a word and a long word.
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. A 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 are not like other reserved words; they 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" to computer scientists).

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 t he 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 in scope.

For recursion, we require dynamic scoping where the most recent active binding is used.

You can use dynamic scoping in place of lexical scoping in non-block-structured languages.

The scope of a name is not necessarily the same as the lifetime of the object to which it is bound (e.g. pointer as parameters).

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

You can also have static local storage where name is scoped but the lifetime of the variable is immortal.

When the program re-enters a scope containing a static variable, it will always point to the same location in memory, thereby saving the value.

The scoping discipline (lexical vs. dynamic) should be part of a formal definition of any programming language.

Names are aka. symbols, and they can be a name of a class, an object, a variable, a method; anything that needs to be referenced. In other words, it is bounded to some object or value. A symbol table is a dictionary of names and their definitions. P

Typeless symbols are symbols that are implemented in all applications of the language. For example, integers and characters are primitive to Java, but strings are not as they are essentially an array of characters. Typeless symbols are names that are not stored into a dictionary and must be set to a value when created. Ruby and Python are good examples of this, where you cannot declare a variable without its value. Binding is the process of creating an association with a value or an object to the name or symbol. There are several different types of binding: Static binding means the variable is fixed at compile-time. Dynamic binding is when the program is running. This is due to the lifetime of the program, otherwise it is said to be a dynamic binding, where the name can be unbound to a value or object. Block structured binding preserves the binding of a variable, such that when a new structure is made with the same name and a different binding, when that structure is closed, the original definition is restored. The original value is restored after executing the method. Some languages allow the programmer to bind a name to a constant or literal value, which cannot be changed. The constant has different scopes in various languages. For example, in C constants must be declared at the top of the page and can be used anywhere. In Java, they can be declared anywhere and be bound to different scopes. The Scope of a name is used to define where the name can be used in a program. In Java, there are global variables that can be used everywhere, but there are also local variables that can only be used within the same class. Scope can be changed by block structures, where variables created within the block (e.g. a function or structure), their scopes are bound to that block. Compilers translate an entire program at once before execution, so the program must be error free. Interpreters instead translate and execute the code one line at a time, until an error is found (Ruby).

Chapter 8:

Binding creates an association between a name and its storage object. Binding can be static; the name is bound to the object when it is allocated and remains bound throughout the program.

Or, binding can be dynamic. In this case, a name can be bound to one object, unboun d, and rebound to another within the run of a program. Names are just as English string used to properly indicate syntax -- more for human ease to read code, since a compiler is fine with any name.

Symbol tables maintain all names and their definitions during run time. Type is stored here as well.

Binding creates an association between a name (in the symbol table) and a storage object (an area of memory). We can picture a binding as a pointer from the name to the storage object.

For dynamic binding (languages with scope) symbol table is called a dictionary and contains name field (name), link field (to organize directory to link to corresponding value), code field (type), parameter field (data).

When there isn't a one-to-one correspondence between a name and object -- it is because use a pointer to storage is used to refer to the data rather than the name. When multiple-name binding is used, storage is not allocated for the second name, but it is bound to the same address as the first and serves as a second way to refer to the same data.

In a constant declaration, the constant value is defined by an expression, that expression will be evaluated once, at compile time. In a macro definition, the expression will be evaluated at run time every time the constant name is used.

The function name is that part of the program in which the name is known and 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.

In a block structured program, a name becomes visible when it is born and invisible when it dies. However, a living name also becomes temporarily invisible when it is masked by a declaration in an inner block, and it becomes visible again when the masking declaration is removed.

If a symbol name to storage location is done within the compiler, even for local variables that become allocated in the middle of execution.

OS: almost all machines run under an operating system (OS). The OS forms an interface between the hardware, the user, and the user's program, and it creates the environment in which the user's program runs. Compiled Language Systems: include APL, FORTH, prolog): These languages are embedded in subsystems that take the place of the OS in forming the user's program de velopment environment.

Chapter 9:

Outside-in: Start evaluating the function body. When (and if) a parameter is used in the body, evaluate the corresponding argument. There are two variants, as follows:

1. Call-by-name: An argument expression is evaluated each time the corresponding parameter is used. This is inefficient and may result in different values for the argument during different times.

2. Call-by-value (strict evaluation): An argument expression is evaluated the first time it is parameterized, and the result is stored for future use.

Indefinite-length argument lists: Parameters might be passed but not named. These can be useful if some way is supplied by the language for referring to them. Several modern languages and systems support optional parameters.

Handle the problem of Parameter Correspondence: Permit the user to use adjacent commas to hold the place* for an omitted argument. Require all arguments that are supplied to precede the ones that are omitted. Use correspondence-by-keyword for all arguments after the first one that is omitted, or for all optional arguments.

Call-by-value: the simplest and cleanest parameter passing mechanism, both in terms of its semantics and its implementation.

Call-by-Name: In the vocabulary of programming languages, lambda calculus uses a call
1-by-name parameter interpretation rule.
Call-by-Reference: The call-by-reference parameter passing mechanism is also called call-by-address and, in Pascal, VAR parameter passing.
Call-by-Return: 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.
Call-By-Value and Call-By-Reference: We can combine call-by-value and call-by-return to achieve a two-way communication between called and caller, with no accessing the restrictions inside the subroutine.
Call-by-pointer: a subcase of call-by-value. The contents of the pointer variable n in the calling program is an address.
A higher-order function: one that takes a function as an argument or returns a function as its result.
Mapping Functions. Frequently, we have a monadic or dyadic operation that we wish to apply to a list or array of data objects.
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.
*** 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 indefinite-length parameter lists, where all parameters are put on a stack as many as you want.
This is easier to write but has potential to crash the system for too much stack space or empty stacks.
2.2 Which is the argument mean?
There are different ways to pass arguments to functions.
Call-by-value: make separate copy in function and work with that.
Call-by-name: make the parameter values for the parameters given. Dangerous as it is a good practice of function passing function to function.
Call-by-reference: make pointer to values that are passed to, this process changes the values but not the pointers.
Call-by-return: able only to change parameter value, not read it.
Call-by-pointer: same as reference but you pass pointer address, this has danger of pointers pointing to null.
Chapter 10.3
Functional Arguments: It is not too difficult and it is very useful to support function arguments as other functions. Let us consider two of the more common application of functional arguments, flexible code and mapping functions.
Implementation: A functional argument can be passed to another function easily and efficiently by passing a pointer to the function, not the function itself. The only implementation problem concerns type-checking—the type of the functional argument to be the types of all its arguments and the type of its result, must be known before code can be compiled to call that function. This accounts for one major difference between the functional languages and Pascal or ANSI C.
Curried functional language. Every functional language has a more arguments is considered to be a higher-order function of one argument that returns another function.
This way of looking at functions is called currying.
Returning Functions from Functions: C supports functional arguments, in the form of pointers to functions, which allow to pass one or more arguments, which are referred to as parameters.
Implementation of Closures: Closures is a useful device for taking general, library code modules and tailoring them to the needs of a particular environment.
*** Chapter 10: Chapter 10.1
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 larger program units. Execution starts when the computer is turned on or receives a RESET signal.
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.
Procedureual languages have two more basic kinds of control: execution of a sequence of instructions and the subroutine.
Subroutine Call: This instruction, also called jump to subroutine, saves the current return address and brings control back to the instruction after the call.
Jump and Conditional Jump: Jump and conditional jump instructions change the next instruction to be executed by storing a new address into the PC. They differ from a jump to subroutine instruction in that they do not save or restore the address from which the call was made.
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 graphical representation of control flow that cannot represent the difference between GOTO and structured transfers of control.
10.2 Conditional Control Structures

- Chapter 10

Branching to that label at the bottom of the loop.

- Mostly used with I/O interrupt system, a variation of infinite loop.
- Conditional (Finite) Loops

1. REPEAT LOOP - Similar to primitive loop with conditional GOTO at the bottom of the scope
   a. Easier and less error prone to translate to GOTO versions
   b. But since it executes at least one, faulty input data can be dangerous
2. WHILE LOOP - Conditional branching is place at the top of the scope
   a. Appropriate for iterative expression/statements
   b. General loops
      - structured loop that merges REPEAT and WHILE
      - <LOOP> - <STATEMENT>WHILE/UNTIL TEST<STATEMENT>-END LOOP>
   Counted Loops
   - The # of iterations and loop exists when iteration count reaches the set #.
   - The # of iteration doesn’t depend on result of computation within the loop
   - <LOOP>[INIT CNT;][COUNTER CHECK<STATEMENT>][COUNTER++<END LOOP>
   - having the counter as a local variable, by nature, makes sense since it is only handled as a local meaning, increase modularity and minimizes error
   - Fixed values before loop (if changed, then the loop is unpredictable)
   - Initial value of [COUNTER]
   - Goal of [COUNTER]
   - Increment value of [COUNTER]
   - Meaning of counter after exiting from loop
   - If exited normally, [counter] = [init] + [stepsizes]
   - If exited with GOTO, [counter] = [counter at exit]

- Iteration Element
  a) Expressions (initializations) to be evaluated before entering the loop, control passes to (b) after execution
  b) The result of the expression/condition must be interpretable as truth value, loop p as long as it’s TRUE
  c) Expressions after the scope of the loop, control passes back to (b)
  - Scope
    - <BEGIN SCOPE> <EXPRESSION> <END SCOPE>
  - Scope begins after the control element
  - The program objects are aggregated as the result of the entire iteration expression
  - Since the iteration element is evaluated every iteration, super-efficient implemenation is not feasible

- Difference between iteration element and ordinary for loops
  - Control element is unrestricted to what kind of work can be done (i.e. scope could be empty)
  - for (current=list, sum=0; current current->next) = NULL; sum += current->value;}

- **Implicit Iteration**

- Only supported in functional languages.

- Motivation: Implicit looping allows processing of all the elements in a data aggregate as a single unit

- Flexibility (any size of arrays can be accommodated, primarily supports the list data type)

- Backtracking
  - Example: if the semantics of a command requires all the possibilities to be examined before returning a FAIL result.
  - Now: Assume SUCCESS, and if the process later fails, the system backtracks to the point it made the trail match, discard that possibility and search for a different trail.

- Implementation: stack and pointers of the positions where early trial match was made.

- Applications: At and string processing require exhaustive searches.

- Chapter 11: Global Control

- Expressions must be remembered during translation so that subsequent labels that are found can be matched against the GOTO’s. This is slow and inefficient.

- Proofs divide programs into “one-in-one-out” sections, but you cannot divide a program between a GOTO and its target label. GOTO’s make correctness and debugging more difficult.

- 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.

- While GOTO’s slow down translation and make code messier, they are also sometimes 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.

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

- A continuation is a “checkpoint” containing everything that is needed in order to safely restart execution at a later time (program counter, stack, environment, etc.) 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 error codes, err or propagation passes exception information to an exception handler, and the handler takes action before resuming the continuation.

- **Continuations**

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

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

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

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

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

- Ignoring exceptions discourages locality of effects.

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

- Software Exception: some languages provide a general exception handling control structure

- Passing Control: When exception occurs it is more useful to propagate error to where it can be handled by passing the exception up the chain of calls.

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

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

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

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

- When the GOTO was first introduced into higher-level languages, it was thought to be both basic and necessary. It was considered basic because it directly reﬂected the branching implicit in all computers because it was used to compensate for the nonexistence of an important semantic mechanism. We can divide the faults of the GOTO control structure roughly into three categories: bad effects on translation, bad effects on proofs of correctness, and bad human engineering properties.

- GOTO is not a problem. The problem is many people who use it.

- Many programmers consider using the GOTO construct bad practice because they think it leads to spaghetti code. A program called SpaghettiCode. This was true at one time, and led to the declaration GotoConsideredHarmful. Called spaghetti code because of its tangled nature. Spaghetti code is easy to write but tricky to debug. The outstanding characteristic of spaghetti code is that virtually everything has global effects. A spaghettified program has a short useful lifetime and poor portability.

- 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. 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 in depth. A variety of attempts has been made to minimize the use of GOTO. 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 anywhere. This is called a normal exit, or FINI SW. On the other hand, if the item is in the data base, a search loop should terminate as soon as it is found, before examining all the remaining items. This is called an abortive exit, or ABORT. In addition to statement labels, for use with the GOTO instruction, Ada also has loop labels, which are used to create double-exit loops are used. At the top of the loop, the step exists. When the loop label takes control immediately out of the enclosing control structure, and into the
control scope that surrounds it.
An exception is a situation that makes it impossible for a program to proceed with normal processing. It is an unpredictable situation with which the program must cope.

Exceptions happen frequently enough so that a robust system cannot ignore them, but seldom enough so that they are unusual.

There are three major ways in which exceptions arise:
1. A hardware error occurs and triggers a hardware interrupt signal.
2. A system software module identifies an error situation.
3. A user function identifies a logically impossible or inconsistent situation.

****

Chapter 13 - Logic Programming

Predicate Calculus: Formulas in predCal made up of a) constants within the universe of discourse, b) variables, c) functions, d) predicates, e) quantifiers (???, ???) denoting sets of objects. Proving true universally quantified predicates is difficult, Falsity is trivial.

Proof Systems: a set of valid deduction rules applied to axioms to prove theorems.

Classical Logic - based on modus ponens ( A -> B, A) -> B). Limitations: can't handle quantifiers / free variables.

Clausal Logic: classical + resolution deduction rule. Can shorten proofs: one resolution application replaces many modus ponens.

Properties of a logical theory: Complete - if every sentence that is true can be proven from its axioms. Decidable - if one can prove/decide whether a particular sentence is/isn't valid. Model describes theory's semantic intent.

Resolution Theorem Provers: resolution is used to "prune" the tree, avoids combinatorial explosion. Unification: identifies clauses to 'resolve'.

Prolog: facts are unconditional Horn Clauses: HORN: Conc. <-- Prem. Prolog: Conc.: = Prem. Variables: scope local to predicate, but can be passed as arguments by reference. The “_” character is an anonymous variable.

Queries: = GOAL of proof process, a request to prove a theorem. Processing aborted if predicate can't be satisfied.

Backtracking: if 'fail' occurs while processing a rule, recursive descent stops, and Prolog moves up then explores another branch of the DFS tree.

Cuts(!): Like 'not'. Can assure of termination, gives control of backtracking. Unsafe cut destroys completeness. Safe cuts won't cause provable goal to fail.

****

Logic programming computation done by finding data objects that satisfy a set of constraints.

Logic language is declarative not imperative because the programmer does not define nature or exact order of computation.

Pred Calc formulas made of: constants (one specific object), variables (range over domain, rep unspecified object from domain), functions, predicates (symbol denotes property of object(s)), quantifiers (??? and ???), logical operations (and, or, not, implies).

Proof system set of valid deduction rules applied to axioms to deduce theorems. Classical logic modus ponens: A->B A true so B true. Clausal logic based on resolution (used in Prolog).

Resolution shorter proof b/c can take place of many modus ponens.

Model is interpretation of semantic intent of a theory in metalanguage.

Automatic theory proving 3 theorems: predicate calc is complete, if proof is possible mechanical proof method exists, no algorithm for propositions that are not true and cannot be proved.

Resolution Theorem Provers use clausal logic connected by or operators.

Resolution: A->(B or C) & (C or D)->E = (A or D)->(B or E).

Resolution proof is a deduction whose conclusion is false (contradiction).

Prolog uses Horn clause (at most one unnegated predicate symbol). Query is request to prove theorem/goal. Prolog takes each term in goal, recursively satisfies term.

Cut avoids trap in lengthy deduction, but may ruin completeness of proof system.

Safe cuts cannot possibly cause a provable goal to fail. Unsafe cuts may be needed for efficiency.

Prolog performance is limited by its interactive, interpretive nature, lack of destructive operations.

Prolog useful when programmer does not know how to organize data or computational process.