Chapter 1 - The Nature of Language:

Syntax touches on the grammar/symbols - the set of rules, and principles that
govern the structure of sentences in a given language. The semantics specifies
the meaning attached to each placement of a word in a sentence, the meaning of
omitting a sentence element, and the meaning of each individual word. 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. Structure: Programming languages
are structurally strict. Structural rules govern the order of statements and
sections of code and particular ways to begin, punctuate and end every program.
Natural languages are more flexible. 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"). Compilers can’t correct more
than trivial semantic errors, those that attempt more make a tradeoff of
time/space. In English, structural deviations are common but for programming
languages, compilers are not very tolerant of deviations.

Redundancy: In English, very redundant, meaning can be ascertained despite many
effects. Whereas programming languages are partly redundant, and the required
redundancy serves as a way to identify errors: (i.e., overloading functions,
blocks, explicit/duplicate type declarations).

Implicit communication: English - read between the lines. However, when
learning a new language, a programmer 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.

Abstraction: both English and programming languages give us a great deal of
abstractions.

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

Standardized language: 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 relearned
and rewritten for each new dialect.

Chapter 2

- A program is a set of actions (low level) or model of a process (high level)
- Language-supporting worldview: 1) provides an API; 2) abstracts real objects
- Builder langs use specific computer funcs; architect langs use an architecture
- Builder languages are more efficient than architecture languages
- Compilers translate architect languages into builder languages
- A representation of an object is a list of relevant facts about the object
- 3 aspects of quality of representation
  1) semantic intent: what programmer intends code to do; often created by data type
  2) explicit (declared & translated) vs implicit (deduced by statement arrangement)
  3) coherent (1 entity = 1 symbol) vs diffuse (1 entity = multiple symbols)
Implicit information is represented by statement arrangement.

Lang. design goals: utility, convenience, efficiency, portability, readability, modeling ability, modeling ability, simplicity, semantic clarity

Power of restrictions: distinguish between meaningful statements and nonsense

Principles of Evaluating Design:

1) Frequency: frequent features should use convenient, easy to understand syntax
2) Locality: keep effects of code as confined as possible
3) Global Variables: 3 types: unrestricted (all vars), locals (good!), no globals
4) Lexical Coherence: code that belongs together should be physically adjacent
4) Distinct Representation: each semantic idea has one syntactic item
5) Too Much Flexibility: useless features that may cause semantic/syntactic errors
6) Semantic Power: express the model, the whole model, nothing but the model
7) Portability: compilable on many different platforms, limits flexibility

Interactive langs: reads expression, evaluates it, write results (LISP, BASIC)

Structured langs: doesn't need GOTO and does pass-by-value (PASCAL, C)

Strongly-Typed langs: objects are named and given type (ANSI C, PASCAL)

OO langs: nondisjoint types & function inheritance; can have subtypes (C++)

Procedural langs: sequentially evaluated statements, uses variables (C, PASCAL)

Old Functional langs: nested set of expressions and function calls (Scheme)

New Functional langs: values passed by functions & params; call-by-need (Haskell)

Parallel langs: program can perform asynchronously, useful for AI (LINDA)

Special: data (MySQL), logic (PROLOG), business (Ada), string (ICON), cmd (UNIX)

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: Exist without being attached, or bound, to an object (LISP).
* Be bound simultaneously to different objects in different scopes
(ALGOL, Pascal).
* Be bound to different types of objects at different times (APL, LISP).
* Be bound, through a pointer, to an object that no longer exists (C).
Pronouns: Pointers: The most important use of pointers in programming languages
is to label objects that are dynamically created
Adjectives: Data Types: data type attributes that can be associated with an
object by a declaration or by a default
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, used analogously to delimit phrases and denote choices
and repetition (WHILE, ELSE, BY, CASE, etc.)

3.2 (up to comments):
Metallanguage permits a language to talk about itself. ex "the paragraph below"
Lexic Tokens are names, nums, strings, & symbols. They r 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
Putting a symbol (commonly ";") denotes the end of a sentence
Scopes (paragraphs) can be defined with brackets and/or statements (ie. if, for)
3.2 Pt 2
Old languages used specific marks in a fixed position, which limited usefulness
Comment 'overflow' (with block comments) cannot be detected by translator
106: Complete language has single-line, partial-line (//) and block (/**/)) comments
107: Symbol Table or Dictionary- the set of syntactic words the compiler recognizes
108: Preprocessor adds syntax without semantics-you cannot add language functionality
109: Lexical analysis must be done before macro expansion to identify macros,
110: as well as after expansion during compilation.
111: C did not specify if tokenization is done before/after preprocessing. Problem
112: because quoted strings must be one token so that they are not edited inside.
113: ANSI C specified tokenization is done first to solve this problem.
114: ***
115: 3.1 Parts of Speech
116: 3.1.1
117: Nouns
118: Variable declaration:
119: - Setting aside storage to represent a real world object
120: - First Class Objects - Objects that can be manipulated and processed as whole
121: units.
122: - giving the memory a name so that the it can be accessed.
123: In different programming languages a name can:
124: Exist without being bound to an object
125: Be bound to several objects in different scopes simultaneously
126: Be bound to different objects at different times
127: Be bound through a pointer to an object that doesn’t exist
128: 3.1.3 Adjective Data Types
129: Corresponds to attributes that are associated with an object
130: 3.14 Verbs
131: Includes: Functions, Procedures, and Operators
132: Comments
133: Can be added to convey general information about the text or program.
134: 3.2 Metalanguage
135: Assortment of syntax delimiters, metawords, and definitions and ways to refer
136: to structural units.
137: (Lexical tokens, Statements, Scope, Comments)
138: **** Chapter 4
139: 4.3.4 to 4.4
140: Lambda Calculus- formula written in symbols manipulated according to logical rules
141: 2 symbol types: single character (variable) and punctuation ‘(‘ ’)’ ‘.’ and ‘\’
142: Formulas: single variable, F(defined), \y.F (lambda expression), FG (application)
143: Variable directly after \ is bound, any occurrence after is bound to \, else free
144: eta reduction: \x.f(x) -> f where f(x) is a function
145: Reducing: renaming and substituting until it is put into normal form.
146: Extension: set of definitions which augment a language with a new facility that
147: can be used in the same way that preexisting facilities are used (eg new type)
148: Lambda calculus defines a minimal semantic basis for computation. Using variables,
149: lambda expressions, and applications, lambda calculus can represent program
150: computations.
151: - Symbols: single-character ‘y’ (variable), or punctuation’(‘ ’)’ ‘.’ and ‘\’
152: - Rules: ‘y’ (variable), ‘\y.F’ (expression), and ‘FG’ (application) are formulas
153: A language’s semantics are extendable if they may be augmented with new actions,
154: data types, or control structures.
155: ***Syntax***
The syntax of a language is its grammatical rules. These are usually defined through EBNF (Extended Backus-Naur Form) and/or syntax diagrams. The meaning of a program is represented by program code or by a computation tree. The language syntax defines the computation tree that corresponds to each legal source program. 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.

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.

EBNF

1. An EBNF language definition can be translated by a program called a parser generator into a program called a parser.

2. A parser reads the user’s source code programs and determines the syntactic category (part of speech) of every source symbol and combination of symbols.

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.

An EBNF grammar consists of: (**several minor variations exist)

Ex. s ::= ( a | bc ) d <-- this rule indicates a s can be replaced by an 'ad' or a 'bcd'

Parsing a program

The parsing routines of a compiler determine how the source code corresponds to the grammar. The output from the parser is a tree-representation of the grammatical structure of the code called a parse tree.

The two broad categories of parsing algorithms are called "bottom-up" and "top-down".

Top-down parsing --> the parser starts with the grammar’s starting symbol and tries, at each step, to generate the next part of the source code string.

Bottom-up --> the parser searches the source code for a string which occurs as one alternative on the right side of some production rule. Ambiguity is resolved by looking ahead k input symbols. The matching string is replaced by the non-terminal on the left of that rule. By repeating this process, the program is eventually reduced, phrase by phrase, back to the starting symbol.

Syntax Diagram

A syntax diagram definition has the same elements as an EBNF grammar, as follows:

A starting symbol

Terminal symbols, written in boldface but without quotes, sometimes also enclosed in round or oval boxes.

Non terminal symbols, written in regular type.

Production rules are written using arrows (as in a flow chart) to indicate alternatives, options, and indefinite repetition. Each rule starts with a nonterminal symbol written at the left and ends where the arrow ends on the right.

Semantics

Semantics are the rules for interpreting the meaning of programming language statements.

The semantic specification of a language defines how each computation tree is to be implemented on a machine so that it retains its meaning.

**** Chapter 5 -- Primitive Types

Computer Memory: Computer memory are arrays of bits organized into group. These bits are grouped in 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 two’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 aspect of floating point. The sign bit is always at the left end. Floating points also have an exponent and mantissa part.

Data Type: an abstraction of common property 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))

Cost of making a type a primitive type:

Added feature complicates 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.

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

**** Chapter 6 -- Modeling Objects:

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

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

Multiple assignment: write a single expression which assigns a value to several storage objects.

Dereferencing: act of extracting the contents from a storage object.

Language that requires explicit dereferences (such as FORTH) benefits from simplicity, but adds visual clutter.

Pointer assignment: a reference that requires a reference to a pointer variable.

Deallocation: recycling process by which dead storage objects are destroyed.

Static storage objects are allocated before execution begins and are immortal (lives until program termination).

Language with only static storage cannot support recursion.

In a totally unrestricted lifetime pattern, a storage object can be born or...
can die at any time
- In a nested lifetime pattern, the lifetime of one object is completely contained within the lifetime of another
A storage object’s lifetime can coincide with the time that the block it’s in is active
Can be managed using a run-time stack
- The lexical parent of a block is the block which encloses the program listing
- The static link points to the stack frame of the block’s lexical parent
- dynamic parent of a block is the block which called it and to which it must return
- The dynamic link points to the stack frame of the current block’s dynamic parent
- heap-allocated object can be born at any time
- Death of such an object occurs when the last reference to the object is destroyed (natural death) or when it is explicitly killed
- A dead heap object is called garbage
- Explicit killing storage objects allows for its reference to be put on a freelist
- Some languages ignore dead cells

**** Chapter 7
Name(symbols): bounded to class/object/variable/method that anything needs to be referenced. must be given a meaning before it can be used, cannot be created dynamically in most languages.
symbol table(dictionary): a data structure store names and their definitions during translation.
FORTH(interactive, typed lang): permits user to redefine a word that is already in the symbol table.
Primitive symbols: names of the built-in functions, types, and constants and their definitions, do not occur in syntax. Named by language designer(eg. Int; sqrt(); TRUE).
Typeless symbols: names not stored into a dictionary and must be set to a value when created. Examples are Ruby and Python, where you cannot declare a variable without its value.
Binding: process of creating an association with a value or an object to the name or symbol.
1) Static binding: if it never changes during the lifetime of the program, otherwise it is a 2) dynamic binding, where the name can be unbound to a value or object. 3) fully dynamic binding: the type must be stored with the object in memory ("typeless") 4) Block structured binding preserves the binding of a variable, such that when a new structure is made with the same name but different binding, when that structure is closed, original definition is restored. Original value is restored after executing the method.
Some languages allow the programmer to bind a name to a constant or
The Scope of a name is used to define where the name can be used in program. Eg. Java: global variables VS private variables. Scope can be changed by block structures, where variables created within the block (e.g. a method or if structure). Compilers translate an entire program at once before execution (no error allowed). Interpreters instead translate and execute the code one line at a time, until an error is found (Ruby). 

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

- 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 value is immortal.

**** Chapter 8:

Names are just an English string used to properly indicate syntax; for humans ease to read code.
Symbol table is a data structure that maintains all names and their definitions during run time. Type is stored here as well.
Static binding creates an association (similar to a pointer) between a name (in the symbol table) and a storage object (an area of memory).
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 correct name), code field (type), parameter field (data).
When there isn’t a one-to-one correspondence between a name and object it is because 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 storage object.
In a constant declaration, if 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 scope of a name is that part of the program in which the name is known and will be recognized by the translator. Scope can be global (name is known throughout the program), local (it is only known within that program block which defined it).

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 masked by a declaration in an inner block, and it becomes visible again when the masking variable dies.

All binding of symbolic name to storage location is done within the compiler.

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.

Interactive Language Systems (Prolog): These languages are embedded in subsystems that take the place of the OS in forming the user’s program development environment.

Complete program: 1. Def of objects 2. Def of functions (parameter names) 3. expressions or function calls with actual arguments procedural languages (eg: ALGOL, FORTRAN, Pascal, APL)

An operator with two arguments is called binary or dyadic; eg: ()
A single-argument operator, known as unary or monadic; eg: +

(a * ((b - c) + a)) Prefix: * a + - b c a Postfix: a b c - a + *

Inside-out (strict evaluation): Evaluate every argument before beginning to evaluate the function.

Outside-in: Start evaluating the function body. When 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 at different times.
2. Call-by-need (lazy evaluation): An argument expression is evaluated the first time its parameter is used, and the result is stored for future use.

Chapter 9

9.1 Function Syntax:

OPTIONAL PARAMETERS: Some parameters can be left out. Danger of making the code hard to understand but may reduce unnecessary redundant values.
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.

INDEFINITE LENGTH ARGUMENT LISTS: limitess amount of parameters are put on a stack. Easier to write but can crash system if stacks are too large, or are empty.

9.2 What does an argument mean?

CALL-BY-VALUE: Make separate copy in function and work with that.

CALL-BY-NAME: Make the parameter values stand for the parameters given.

- Dangerous as the values inside function can change unexpectedly.

CALL-BY-REFERENCE: Make pointer to values that are passed to, this process changes the values but not the pointers made.

CALL-BY-RETURN: Able only to change parameter value, not read it.

To avoid consuming the execution time and stack space necessary to pass a very large argument by value.

CALL-BY-VALUE-AND-RETURN: Make copy and return updated copy. Two-way communication
between called and caller, with no accessing restrictions inside the subroutine.

CALL-BY-POINTER: Same as reference but you pass pointer address, this has danger of pointers pointing to null.

Chapter 9.3

FUNCTIONAL ARGUMENTS: - Supports functions as arguments to other functions.
- Flexible code and mapping functions are applications of functional arguments.

IMPLEMENTATION: - A functional argument can be passed to another function by passing a pointer to the function, not the function itself. The only concern is type checking - the argument type(s) and return type, must be known before the code can be compiled to call that function. This is one major difference between the functional languages and Pascal or ANSI C.

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. This way of looking at functions is called currying.

RETURNING FUNCTIONS FROM FUNCTIONS: C supports functional arguments, in the form of pointers to functions, and C++ carries this further by permitting predefined operators to be named and used as arguments as well. Both languages permit the programmer to return a pointer to a function as the result of a function. However, a C or C++ function cannot create a function within a function and return it as the result of the function.

CLOSURES: Closure is an operation that can be applied to an expression. We close 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 process. An example of this would be partial parameterization.

IMPLEMENTATION OF CLOSURES: Closure is a useful device for taking general, library code modules and tailoring them to the needs of a particular environment.

*** Chapter 10 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. Machine instructions are executed in the order in which they are loaded into memory unless that order is changed by a jump instruction.

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. Subroutine 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. 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 originates.

Conditional expressions return some result, such as b< ?b: c; however, conditional statement returns no results; also, they can be transformed to each other. We can improve the lexical coherence of IF-GO by emulate a structured IF THEN ELSE. For
structured conditionals, we can use THEN...ELSE, which makes programs have fewer keywords and language syntax simpler but causes ambiguity in the dangling else situation, and we can also use THEN...ENDIF, which make programs not filled with BEGIN...END pairs and solves the ambiguity problem. Elseife 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 the program will continue to next label without breaking.

Infinite loop is used to implement processes that interact by using the IO interrupt system. REPEAT is always executed at least once. A While test loops on TRUE, and a Until test loops on FALSE. Restricting loops to single exit may cause priming read -- a read instruction given to bring in the first line of data before entering the main processing loop. A general loop combines WHILE 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 entry, otherwise the number of iteration is unpredictable. The most general form of the FOR loop is called the iteration element, and C FOR loop is not a counted loop. The control element of FOR contains a list of expression to be evaluated before entering the loop, an expression whose result must be interpreted as truth table and an increment after the scope of the loop. Implicit looping in functional language 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 trial match was made.

**** Chapter 11: Global Control:

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 branch instructions of all computers. It was considered necessary 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.

Many programmers consider using the GOTO construct bad practice because they think it leads to something 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 spaghetti program has a short useful lifetime and poor portability.

Many of the newer computer languages contain no GOTO at all.

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 outermost loop.

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 an exception handler, and the handler takes action before resuming the continuation.

-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.

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 then 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 error.

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

**** Chapter 13 - Logic Programming ****

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

Logic language: declarative not imperative, programmer does not specify nature or exact order of computation.

Predicate Calculus: Formulas in predCal made up of a) constants within the universe of discourse (a specific object), b) variables (range over domain, repr. unspecified object from domain), c) functions, d) predicates (symbol denoting property of object(s)), e) quantifiers (there exists, for all) denoting sets of objects. f) operators (&, v, ->, etc.)

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, therefore B).

Limitation: 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 is one interpretation of the semantic intent of a theory in metalanguage.

Automatic theory proving - 3 theorems: 1) Predicate calc complete 2) if proof is possible mechanical proof method exists 3) no algorithm for propositions that are not true and cannot be proved.

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

Resolution Theorem Provers use clausal logic connected by or operators.

Resolution example: 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 facts are unconditional. Uses Horn clause- at most one unnegated predicate.
574: Variables: scope local to predicate, BUT can be passed as arguments/by reference.
575: "_" character is an anonymous variable.
576: Query: is the GOAL of proof process, a request to prove a theorem. Processing
577: is aborted if a predicate can’t be satisfied.
578: Backtracking: if 'fail' occurs while processing a rule, recursive descent stops,
579: and Prolog moves up then explores another branch of the DFS tree.
580: Cuts(!): Like 'not'. Can assure of termination, gives control of backtracking,
581: prevents traps. Unsafe cut destroys completeness. Safe cut won’t cause provable
582: goal to fail.
583: Prolog performance: limited by its interactive, interpretive nature, and lack of
584: destructive operations
585: Prolog useful if programmer unsure how to organize data or computational process.