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 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 these languages into machine language. 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 corresponds to the rules of the language, not 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: for programming languages, they are not. Redundancy: In English, very redundant, meaning can be ascertained despite many errors. Whereas programming languages are partly redundant, and the required redundancy serves as a way to identify errors: [e.g., overloaded functions, blocks, syntax diagrams, 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. 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. If this happens, we say a program is portable. 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 4

1.3.4 to 4.4

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

Formulas:

\[ \text{formulas} \] symbol types: single character (variable) and punctuation (', ' , ) and '(' \\
\[ \text{Formulas: \; \text{single \; variable, \; F(defined), \; \gamma . F (lambda \; expression), \; FG (application) \; \text{variable \; directly \; after \; \gamma \; is \; bound, \; any \; occurrence \; after \; \gamma \; is \; bound \; to \; \gamma, \; else \; free \; eta \; destructible } \} \rightarrow \} \rightarrow \] null

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 previous facilities are used (e.g., new types).

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

- Syntax:
  - single-character 'y' (variable), or punctuation (', ' , ) and '(' , ')
  - Rules: 'y' (variable), 'y?y.F' (expression), and 'FG' (application) are formulas

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

Syntactic:

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.

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

-ex. s := ( | a | b | c | d | e ) --- 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" parsing.

Bottom-up 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 of the starting terminals of the rules. Ambiguity is resolved by looking ahead input symbols. The matching string is replaced by the nonterminal on the left of that rule. By repeating this process, the program is eventually reduced, phrase by phrase, back to the starting symbol.

Syntax Diagrams:

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 a left-to-right sequence. Each rule starts with a nonterminal symbol written at the left and ends where the arrow ends on the right.

Semantics:

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 are grouped in 8 to form a byte. 2 and 4 bytes are called a word and a long word.

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.

Floating Point: The IEEE has a standard for floating-point representation and computing facilities that 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 their program representation. Such representation 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 point constructors.))

Cost of making a type a primitive type:

Added feature complicates the language syntax and semantics

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 signatures may be unable to support the type with user implementation.

Built in functions and comparison work only with primitive types.
are not as convenient or easy to use

Chapter 8:
Names are just an English string used to properly indicate syntax; for humans easy 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 a given binding (languages with scope) 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 becomes necessary to store the data under two or more names somewhere.
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 macro definition, 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 found. The current implementation of the translator (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.
Intermediate Language Systems (Prolog, LISP) that take the place of the OS in forming the user’s program development environment.

Example program:
1. Def of objects 2. Def of functions 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; eq: (a+b) is a + b in APL.
A function is a procedure with no storage and a global (name is known and will be found).

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

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

Outward evaluators: 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 as the subroutine is called.
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 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 procedures. In compilers, 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 that they are loaded into the memory. 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 chapter 10). Subroutine call with parameters and conditional execution (discussed in chapter 10).
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 call restores the saved address and brings control back to the instruction after the call. This methodology, of decomposing a program 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 0 or 1; however, conditional statement returns no results; also, they can be transformed to each other. We can improve the lexical coherence of IF-DO by emulating a structured IF THEN ELSE. For structured conditionals we can use THEN...ELSIF...ELSIF pairs and solves the ambiguity problem.Elseif statement can be replaced with symbolic and conditional 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 interface to system. Repeat loop is also used at least once. A While test on TRUTH and a Until test 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 the loop test. For example, in the IF loop, if the condition, 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 found. The current implementation of the translator (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.

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.

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.

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.
- 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.