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: communicating among computers and among humans. It must exist without being attached, or bound, to an object (LISP).

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

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

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

Chapter 3 - Elements of Language

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. Non-Objects: cannot exist without being attached, or bound, to an object (LISP).

2. 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 to delimit phrases and define choices and repetition (WHILE, ELSE, BY, CASE, etc.)

3.2 up to comments:

Standard: putting a language to talk about itself. ex: "the paragraph below" Lexicon Tokens are names, numbers, strings, and symbols. They usually separated by spaces but not always separated by spaces. ex: "bar=foo+35.5" has 6 tokens and no spaces.

Comments: the first step of the compiler. It breaks text into Lexicon Tokens. Putting a symbol (commonly ";") denotes the end of a sentence.

4.3.4 to 4.4 Lambda Calculus: a formula written in symbols manipulated according to logical rules.
2 symbol types: single character (variable) and punctuation '(' ')' '.' and '\'

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

Variable directly after \ is bound, any occurrence after is bound to \, else free

type reduction: \!x.(f(x)) \rightarrow 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 (eq new type)

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), ‘\gamma.F’ (expression), and ‘FG’ (application) are formulas

A language’s semantics are extendable if they may be augmented with new actions, data

types, or control structures.

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 source code program 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 <- all 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 grammatical structure. 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 --- 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 parsing: the parser searches the source code 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 (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 Types: 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 domain 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 the language 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

- Coherant 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

- Dereferncing: 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

- Dynamic allocation: 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
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 block link to the stack frame of the current 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

  free list

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 language): 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 bound 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 literal value, which cannot be changed. eg. C: constants must be declared at the top of the page and can be used anywhere. Java: can be declared anywhere and be bound to any scope. The Scope of a name is used to define where the name can be used in program. E.g., 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...else). For compilers translate an expression 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: Limitless 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-REFERENCE: - The parameter values stand for the parameter values 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-- In and Return: Make copy and return updated copy. Two-way communication between called and caller, with no accessing restrictions inside the subroutine.

FUNDAMENTAL 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 to the function itself, not as object. 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 la., every function has two or more arguments and 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, returning a function cannot return 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, thereby creating a function out of 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 statements. Execution changes the order in which they are evaluated, which 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.

Assemblers: Two kinds of control structures above the expression level are basic: subroutine call with parameters and conditional execution (diagnosing if-then-else). Processors: Two 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 after the call method. By decomposing a program into clearly defined parts with limited and clearly defined interactions, then building a program by impromptu calling of subroutines, the programmer is able to be the basis of good program design.

Jump and Conditional Jump: Jump and conditional jump instruction changes 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 bcc/bcc; however, conditional statement returns no results; also, they can be transformed to each other. We can improve the lexical coherence of IF-THEN-ELSE code emulated IF THEN ELSE. FOR structured programming block, we can use BEGIN...END pairs and solve the ambiguity problem. Elseif statement can be replaced by nested if. Case statement is always evaluated by the first case that matches 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 will never terminate. 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 processing. Processing, therefore, is a REPEAL 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 loop is called the counter or loop, which always has a predefined or not predefined 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 after the selection dependent on the scope of the local definition. A loop 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 a useful tool, but was considered bad programming practice. This led to the branch instructions of all computers. It was considered necessary because it was used to compensate for the nonexistence of a jump in the semantic mechanism. We can divide the faults of the GOTO control structure roughly into these 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 Spaghetti Code. 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. As a result, spaghetti code is not portable and hard to update.

Many of the newer computer languages contain no GOTO at all. Many looping situations require both finishing naturally and aborting early. The BREA keyword is simply a GOTO statement targeting the next line of code out side the loop. The loops, PAR_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. 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 control structures with the exception from 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. How: several languages use 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. OS stores the saved address and generates an interrupt signal which is processed by OS. OC will then set status flags.

Software Environment: Some languages provide a general exception handling control structure. Programming errors.

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 in some cases you 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.

Handling error by 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 languages: declarative not imperative, programmer does not specify name or exact order of computation.

Predicate Calculus: Formulas in preCal made up of a) constants within the universe of discourse (a specific object), b) variables (range over domain, repr. universal quantification), c) predicates (symbols denoting properties of object(s)), d) quantifiers (there exists, for all) denoting sets of objects, e) operators (+, *, \ or \ , etc.)

Proving true universally quantified predicates is difficult. Falsity is trivial. This is called the completeness of the system. If we can prove an axioms we can conclude any other formula.

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 \rightarrow (B or C) & (C or D) \rightarrow E = (A or D) \rightarrow (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
Horn Clauses vs. Prolog: Horn- Conc. \leftarrow Prem. Prolog- Conc. \rightarrow Prem.
Variables: scope local to predicate, BUT can be passed as arguments/ by reference.
"_" character is an anonymous variable.
Query: is the GOAL of proof process, a request to prove a theorem. Processing
is aborted if a 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,
prevents traps. Unsafe cut destroys completeness. Safe cut won’t cause provable
goal to fail.
Prolog performance: limited by its interactive, interpretive nature, and lack of
destructive operations
Prolog useful if programmer unsure how to organize data or computational process.