Abstract Syntax, concrete Syntax, and Pragmatics

- The abstract syntax of data is its structure described as a data type (possibly, but not necessarily, an abstract data type), independent of any particular representation or encoding.

- This is particularly used in the representation of text in computer languages, which are generally stored in a tree structure as an abstract syntax tree.

- Abstract syntax, which only consists of the structure of data, is contrasted with concrete syntax, which also includes information about the representation. For example, concrete syntax includes features like parentheses (for grouping) or commas (for lists) which are not included in the abstract syntax, as they are implicit in the structure.
• Abstract syntax: what are the significant parts of the expression?
• The abstract syntax of an implementation is the set of trees used to represent programs in the implementation. This is, the abstract syntax defines the way the programs look like to the evaluator/compiler.

• Example: a sum expression has its two operand expressions as its significant parts.

• The abstract syntax is part of the definition of a particular implementation (evaluator or compiler) of a language.
Concrete syntax: what does the expression look like?

the concrete syntax is part of the definition of the language.

The concrete syntax of a programming language is defined by a context free grammar. It consists of a set of rules (productions) that define the way programs look like to the programmer.

Example: the same sum expression can look in different ways:

- 2 + 3 -- infix
- (+ 2 3) -- prefix
- (2 3 +) -- postfix
- bipush 2 -- JVM
  bipush 3
  iadd
- the sum of 2 and 3 -- English
• Pragmatically it is related to error level.
• If we need to add more statements, we need to add them within the brackets representing particular group.
• Concrete structure for while loop in C and Pascal is shown below:

```c
while(x!=y)
{
    ...... 
}
```

```pascal
while x<>y do begin
    ...... 
end
```
Semantics

- Semantics is the field concerned with the rigorous mathematical study of the meaning of programming languages. It does so by evaluating the meaning of syntactically legal strings defined by a specific programming language.

- The relationship between syntax and model of computation is given by semantics of programming languages.

- Semantics describes the processes a computer follows when executing a program in that specific language. This can be shown by describing the relationship between the input and output of a program, or an explanation of how the program will execute on a certain platform, hence creating a model of computation.

Symantics : syntax --> computational model

- Above approach is called as syntax directed semantics.
The field of formal semantics encompasses all of the following:

- The definition of semantic models
- The relations between different semantic models
- The relations between different approaches to meaning
- The relation between computation and the underlying mathematical structures from fields such as logic, set theory, model theory, category theory, etc.
There are several approaches used to describe semantics of programming languages. Main / broad classes are:

- Axiomatics
- Denotational
- Operational.

Most variation in formal semantic systems arose from the choice of supporting mathematical formalism. Those variations to describe semantics are:

- Algebraic semantics
- Action semantics
- Attribute semantics
- Categorical
- Concurrency semantics
- Game Semantics
- Predicate transformer semantics
• **Denotational semantics** - whereby each phrase in the language is interpreted as a denotation, i.e. a conceptual meaning that can be thought of abstractly. For example, denotational semantics of functional languages often translate the language into domain theory. Denotational semantic descriptions can also serve as compositional translations from a programming language into the denotational metalanguage and used as a basis for designing compilers.

• **Operational semantics** - whereby the execution of the language is described directly (rather than by translation). Operational semantics loosely corresponds to interpretation, although again the "implementation language" of the interpreter is generally a mathematical formalism. Operational semantics may define an abstract machine (such as the SECD machine), and give meaning to phrases by describing the transitions they induce on states of the machine. Alternatively, as with the pure lambda calculus, operational semantics can be defined via syntactic transformations on phrases of the language itself;
- **Axiomatic semantics**, whereby one gives meaning to phrases by describing the logical axioms that apply to them. Axiomatic semantics makes no distinction between a phrase's meaning and the logical formulas that describe it; its meaning is exactly what can be proven about it in some logic. The canonical example of axiomatic semantics is Hoare logic.

- **Action semantics** is an approach that tries to modularize denotational semantics, splitting the formalization process in two layers (macro and microsemantics) and predefining three semantic entities (actions, data and yielders) to simplify the specification.

- **Game semantics** uses a metaphor inspired by game theory.

- **Categorical** (or "functorial") semantics uses category theory as the core mathematical formalism.
For a variety of reasons, one might wish to describe the relationships between different formal semantics. For example: To prove that a particular operational semantics for a language satisfies the logical formulas of an axiomatic semantics for that language. Such a proof demonstrates that it is "sound" to reason about a particular (operational) interpretation strategy using a particular (axiomatic) proof system.

It is also possible to relate multiple semantics through abstractions via the theory of abstract interpretation.
References and Unnamed Variables

- In C++, anonymous objects are primarily used either to pass or return values without having to create lots of temporary variables to do so. Memory allocated dynamically is also done so anonymously (which is why its address must be assigned to a pointer, otherwise we’d have no way to refer to it).

- However, it is worth noting that anonymous objects are treated as rvalues (not lvalues, which have an address). This means anonymous objects can only be passed or returned by value or const reference. Otherwise, a named variable must be used instead.

- It is also worth noting that because anonymous objects have expression scope, they can only be used once. If you need to reference a value in multiple expressions, you should use a named variable instead.
1.

```cpp
#include <iostream>

int add(int x, int y)
{
    return x + y; // an anonymous object is created to hold and return the result of x + y
}

int main()
{
    std::cout << add(5, 3);
    return 0;
}
```
2.

```cpp
void printValue(int value)
{
    std::cout << value;
}

int main()
{
    PrintValue(5 + 3);   //instead of PrintValue(var), we passed 5+3
    return 0;
}
```
SIMPLESEM is a simple abstract processor. Semantics of programming languages can be described operationally – that is, by describing the language constructs by translating them into a sequence of equivalent SimpleSem instructions. It comprises of Instruction Pointer (reference for current instruction), a memory and a processor. Memory is the place used to store instruction and information to be controlled.
• This machine operates in the following manner:
  while not(halt)
    get the current instruction (C[IP])
    increment IP
    execute the current instruction
  end
• (halt is a special instruction)
• We make use of the notations D[X] and C[X] to represent the values stored in the X-th cell of D and C, resp. X is l_value and D[X] is the subsequent r_value.
• SIMPLESEM will also be improved as new programming language concepts are set up.
Fig. SIMPLESEM MACHINE
Set instruction

- General format: set target, source
- For example
  
  Set 12, D[8]
  
  means “store the content of D[8] into D[12]”
- This is not to be confused with:
  
  set D[12], D[8]
  - which means “store the content of D[8] into D[ whatever is the content of D[12] ]”
  - indirect addressing
- We can combine values into expressions:
  
Reading and writing, Branching (jumps)

- Reading and writing is done via the set instruction:
  - `set 12, read`
- Reads the input value from the keyboard
  - `set write, D[12]`
- Writes the output value onto the screen
- Unconditional jumps
  - `jump 42`
  - means "jump to instruction 42"
- Conditional jumps
  - means "jump to instruction 42 if the condition is satisfied"
Run time Structure

- languages can be classified in several categories, according to their execution-time structure.
- We will proceed gradually, from the most basic concepts to more complex structures that reflect what is provided by modern general-purpose programming languages

a) Static languages:
- Exemplified by the early versions of FORTRAN and COBOL
- these languages guarantee that the memory requirements for any program can be evaluated before program execution begins
- all the needed memory can be allocated before program execution.
- Recursion is not allowed
- Languages of variant C1(simple statements only), C2 and C2'(adding simple methods) falls under this category
b) Stack-based languages:

- Historically headed by ALGOL 60 and exemplified by the family of so-called Algol-like languages.
- This class is more demanding in terms of memory requirements, which cannot be computed at compile time.
- Memory usage is predictable and follows a last-in-first-out discipline: the latest allocated activation record is the next one to be deallocated.
- It is therefore possible to manage SIMPLESEM’s D store as a stack to model the execution-time behavior of this class of languages.
- An implementation of these languages need not use a stack: deallocation of discarded activation records can be avoided if store can be viewed as unbounded.
- The stack is part of the semantic model we provide for the language; strictly speaking, it is not part of the semantics of the language.
- Languages under class 3 (C3-recursion supporting language) and class 4 (C4-support block structure)
c) Fully dynamic languages:

- These languages have an unpredictable memory usage; i.e, data are dynamically allocated only when they are needed during execution.
- Problem due to dynamic memory: Efficient memory management
- In particular, how can unused memory be recognized and reallocated, if needed. To indicate that store D is not handled according to a predefined policy (like a FIFO policy for a stack memory), the term “heap” is traditionally used.
- Language of class C5 (more dynamic behaviors) falls in this category.

- Assignment 2: References and Unnamed variables.
- Assignment 3: Explain Semantics and Its classes.
- Assignment 4: Explain Run time structure of C.