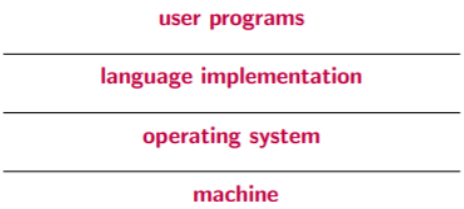
Definition of a programming language

A language has three main components

1. **Syntax:** defines the form of programs, how expression, commands, declarations are built and put together to form a program
2. **Semantics:** gives the meaning of programs; how they behave when they are executed
3. **Implementation:** a software system that can read a program and execute it in a machine, plus a set of tools (editors, debuggers, etc)

Computer Architecture

1. Memory
2. Registers
3. Processor, with a set of machine instructions for arithmetic and logic operations (machine language)



The operating system supplies higher-level primitives than those of the machine language. Language implementations are built on top of these. User programs form the top layer

Implementing a Programming Language

We are concerned with high level programming languages which are (more or less) machine independent. Such languages can be implemented either by:

1. ***Compilation –*** compiling the program into machine code
2. ***Interpreting –*** programs
3. ***A Hybrid Method –*** which combines **compilation** and **interpretation**.

Compilation

The compiler translates source language into machine language that can be executed directly on the computer.

Advantage: This method provides fast program execution.

This process is made of four steps: In which first the source language is tokenized into tokens, which are then parsed by the syntax analyser to check the syntax and produce correct abstract syntax tree, when abstract syntax tree is created the Semantics and Type analysis check's typing, semantics and optimizes the code (some compilers do). Lastly the machine code generation step in which the abstract syntax tree is used to generate machine language code.

1. Lexical analysis
2. Syntax analysis
3. Type and Semantics Analysis
4. Machine code Generation

Execution

The execution of machine code occurs in a process called fetch-execute cycle.

Each machine instruction to be executed is moved from the memory where programs reside, to the processor.

- Initialize program counter (in a register of the machine)
- Repeat forever:
  - Fetch the instruction pointed by the program counter
  - Increment the program counter
  - Decode and execute the instruction

When all instructions in the program have been executed, control returns to the operating system.

Interpretation

The program is executed by an interpreter which simulates a machine whose fetch-execute cycle deals with high-level program instructions.

In this case there is no translation/ generation of code. The interpreter produces a virtual machine for the language.

- Advantages:
  - Easy to implement
  - Easy debugging of programs
  - Better error messages
- Disadvantages:
  - Slow execution

Simpler languages, and prototyping languages are usually interpreted

Hybrid Implementation

Some languages are implemented by compilation to and intermediate language instead of machine language. This intermediate language is then interpreted.

- Advantages:
  - Portability
  - Programs can be executed in any machine that has an interpreter for the intermediate language. (e.g. in Java the intermediate form (bytecode) provides portability to any machine that has a bytecode interpreter)

Syntax

The syntax is concerned with the form of programs

It is given by:

- An **alphabet:** the set of characters that can be used
- A set of **rules**: indicating how to form expressions, commands etc…

We distinguish between ***concrete syntax*** and ***abstract syntax***

- **Concrete Syntax:** describes which chains of characters are well-formed programs.
- **Abstract Syntax:** describes the syntax trees, to which a semantics is associated.

Grammar

To specify the syntax of a language we use *grammars*.

A grammar is given by:

- An alphabet V = which is a union of terminals and non-terminals Vt u Vnt.
- Rules
- Initial Symbol

Arithmetic expressions EXAMPLE

Concrete syntax:

$$\begin{array}{lll} \mathit{Exp} & ::= & \mathit{Num} \mid \mathit{Exp}\ \mathit{Op}\ \mathit{Exp} \\ \mathit{Op} & ::= & + \mid - \mid * \mid \mathit{div} \\ \mathit{Num} & ::= & \mathit{Digit} \mid \mathit{Digit}\ \mathit{Num} \\ \mathit{Digit} & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

Problem: Ambiguity.
How is 1 - 2 - 3 read?

Abstract Syntax:

$$\begin{array}{lll} e & ::= & n \mid \mathit{op}(e, e) \\ \mathit{op} & ::= & + \mid - \mid * \mid \mathit{div} \end{array}$$

This grammar defines **trees**, not strings.

Remarks

- *Abstract syntax describes program representation*
- *Abstract syntax is not ambiguous*
- *We will always work with the abstract syntax of the language and study the semantics of the main constructs of programming languages.*

Semantics

The semantics of a language define the meaning of programs, that is, how they behave when they are executed on a computer.

Different language has different syntax and different semantics for similar constructs, but variations in syntax are often superficial. It is important to appreciate the differences in meaning of apparently similar constructs.

For example, two languages may have the same syntax, but there is a variation in semantics, this will make huge difference.

There are two types of Semantics:

- ***Static Semantics or Typing:*** The goal is to detect (before the actual execution of the program) programs that are syntactically correct but will give errors during execution.
  - ***Example:*** *1 + 'a' or in Python 3/'three*
    '
- ***Dynamic Semantics or just Semantics:*** Specifies the meaning of programs

Informal vs Formal Semantics

Informal definition often given by English explanations in language manuals, are often IMPRECISE and INCOMPLETE.

Format semantics are important for:

- ***The implementation of the language:*** The behaviour of each construct is specified, providing an abstraction of the exaction process which is independent of the machine.
- ***Programmers:*** a formal semantics provides tools or techniques to reason about programs and prove properties of programs. (If we are checking critical algorithms we sometimes need to prove all properties)
- ***Language designers:*** a formal semantics allows to detect ambiguities in the constructs and suggest improvements and new constructs (e.g. influence of the study of the A-calculus in the design of functional languages).

However, formal semantics descriptions can be complex, so usually only a part of language is formally defined.

Formal Semantics – compilers writers, studying language in depth.

Approach of formalizing Semantics / Style of Semantics

tools to describe formal semantics

- **Denotational Semantics:** BUILDS A MATHEMATICAL MODEL IN TERMS OF FUNCTIONS
  - The meaning of expressions (and in general, the meaning of the constructs in the language) is given in an abstract, mathematical way (using a mathematical model for the language). The semantics describes the effect.
- **Axiomatic Semantics:** DESCRIBE PROBLEM AT THE START AND THE END
  - Uses axioms and deduction rules in a specific logic. Predicates or assertions are given before and after each construct, describing the constraints on program variables before and after the execution of the statement (precondition, postcondition).
- **Operational Semantics:** THE MEANING OF EACH CONSTRUCT IS GIVEN IN TERMS OF COMPUTATION STEPS.
  - The behaviour of the program during execution can be described using a transition system (abstract machine – theoretical representation of computer, structural operational semantics – every computation that can take place on the abstract machine).

OPERATIONAL SEMANTICS is very useful for the implementation of the language and for proving correctness of compiler optimizations.

DENOTATIONAL SEMANTICS and AXIOMATIC SEMANTICS are useful to reason and prove properties of programs.

Operational Semantics: Transition Systems

A transition system is specified by:

- A set *Config* of *configurations* or *states*,
- A binary relation $\rightarrow \subseteq Config \times Config$, called *transition relation*.
  We use the notation $c \rightarrow c'$ (infix) to indicate that c, c' are related by $\rightarrow$.

Notation:
$c \rightarrow c'$ denotes a *transition* from c to c' (change of state).
$c \rightarrow^* c'$ is the reflexive transitive closure of $\rightarrow$. In other words, $c \rightarrow^* c'$ holds iff there is a sequence of transitions:

$$c \rightarrow c_1 \rightarrow \ldots \rightarrow c_n = c'$$

where $n \geq 0$.

- Arrow with * means zero or more steps.

Transition Systems

We will distinguish an *initial* and *final* (also called *terminal*) subset of configurations, written I, T. For all $c \in T$ there is no c' such that $c \rightarrow c'$.
The idea is that a sequence of transitions from $i \in I$ to $t \in T$ represents a run of the program.

A transition system is **deterministic** if

$$\text{for all } c, c_1, c_2 : c \rightarrow c_1 \text{ and } c \rightarrow c_2 \text{ implies } c_1 = c_2$$

When transition system is deterministic it has only single possibility for next step, when there are non-deterministic have multiple choices to chose next step.

Abstract Machines

Abstract machines are very useful for implementing a programming language since they describe the execution of each command.

However, for users of the language they are not always easy to understand.

It is possible to specify the operational semantics of the language in a more structured way.

The structural approach to operational semantics based on transition systems gives an inductive definition of the execution of a program.

Each command is described in terms of its components