# Capitolo 1 Introduzione

## 1.1 Motivazione

In un periodo di profonda incertezza come quello che stiamo attraversando a causa della diffusione dell’epidemia di COVID-19, sorge spontaneo interrogarsi sul futuro.   
È in momenti come questi che si sente più forte la necessità di rinnovarsi, sia dal punto di vista individuale che da quello della collettività.

Un ruolo di primo piano all’interno di un processo di rinnovamento della società deve necessariamente essere assunto dall’informatica.  
La pandemia ha dimostrato, se ancora ce ne fosse bisogno, quanto essa sia fondamentale per una realtà moderna e dinamica.  
Ha assunto in questi mesi un ruolo determinante in attività fondamentali per la società quali il lavoro (*smart working*) e la scuola (*Didattica A Distanza*).E le persone, quali attori principali di questo processo, si sono scoperte capaci di utilizzare strumenti come *smartphone*, *tablet* o *PC* senza particolari problemi.  
Molti di questi hanno imparato da adulti ad utilizzare queste tecnologie, altri, invece, sono i cosiddetti *digital natives* [1].   
Proprio quest’ultimi hanno avuto l’opportunità di crescere a stretto contatto con le tecnologie informatiche e, quindi, possiedono conoscenze e competenze di gran lunga superiori a quelle dei propri coetanei, per esempio, di dieci anni fa.  
Per questa ragione ritengo anacronistico e superfluo un processo classico di alfabetizzazione informatica dei più giovani.

## 1.2 Il progetto in breve

In questo lavoro verrà presentato un nuovo linguaggio di programmazione, il *DLK* (*Didactical Language for Kids*).  
Questo linguaggio si offre come un invito alla programmazione di alto livello rivolto ad un pubblico giovane, grazie all’utlizzo di costrutti semplificati, rispetto ai principali linguaggi di programmazione, e di *keywords* in italiano.

Oltre alla specifica del *DLK*, in questa tesi verrà presentata anche la sua implementazione attraverso un interprete.

## 1.3 Digital Natives

Con il termine *digital natives* (nativi digitali, in italiano) si definiscono tutti quegli individui che sono nati nel periodo di diffusione di massa delle tecnologie digitali, quali i *PC* a interfaccia grafica, i telefoni cellulari, internet, ecc.  
Questa esprossione è stata coniata da Mark Prensky nel 2001 ed è stata diffusa in Italia da Paolo Ferri nel 2011 [2].

Secondo Prensky, i nativi digitali sono tutti i nati dal 1985 in poi negli Stati Uniti d’America, mentre in Italia i cosiddetti nativi digitali *puri* vengono identificati nei nati dal 2000 in poi [3].

## 1.4 Introduzione ai linguaggi di programmazione

Un linguaggio di programmazione è uno strumento di astrazione che permette di specificare computazioni tali da poter essere eseguite su un elaboratore.

## 1.4.1 Cenni storici sui linguaggi di programmazione

Essi furono introdotti per la prima volta nel 1837 da Ada Lovelace che sviluppò un linguaggio in grado di far calcolare alla *macchina analitica* di Charles Babbage i numeri di Bernoulli [4].

Successivamente, durante la Seconda guerra mondiale, Konrad Zuse ideò quello che viene individuato come il primo linguaggio di programmazione ad alto livello: il Plankalkül.   
Esso conteneva istruzioni di assegnamento, salti condizionali, cicli di iterazione, array, gestione delle eccezioni, ecc.   
Tuttavia l’implementazione di questo linguaggio risale solamente al 2000, quando presso la Technische Universität Berlin venne scritto il relativo compilatore [5].

Con l’avvento dei primi calcolatori elettronici digitali fece la sua comparsa l’*Assembly*, un linguaggio fortemente legato all’*hardware* dell’elaboratore e molto vicino al *linguaggio macchina*.   
Esso introdusse il concetto di *compilatore*, in quanto *Assembly* si poneva ad un livello di *astrazione* superiore rispetto all’*hardware* e per questa ragione necessitava di essere tradotto in *linguaggio macchina* per essere eseguito, come mostrato in Figura 1.1.  
Il *compilatore Assembly* viene spesso chiamato *Assembler* (*Assemblatore*).

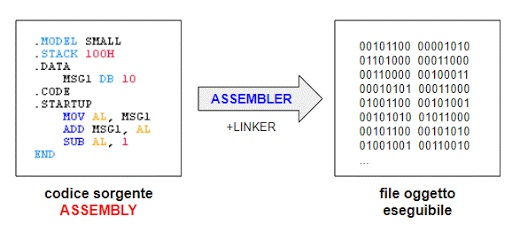


Figura ..1: Traduzione in *linguaggio macchina* da *Assembly*  
©andreaminini.com

Successivamente si cercò di aumentare sempre di più il livello di *astrazione* dei linguaggi fino a giungere, nel 1957, alla specifica del *FORTRAN* ad opera di John Backus.  
Esso fu seguito da altri linguaggi *imperativi* quali: *BASIC* (1964), *Pascal* (1970), *C* (1972), ecc.

Si iniziò, inoltre, ad introdurre nuovi *paradigmi di programmazione* differenti dal classico *paradigma imperativo*.  
Nel 1967, con *Simula* venne introdotto il concetto di *programmazione orientata agli oggetti*, mentre nel 1959 con *Lisp* venne introdotta la *programmazione funzionale*.  
Menzione particolare va fatta anche per *Prolog* che, nel 1972, introdusso il *paradigma di programmazione logico*.

Avvicinandosi ai giorni nostri, vanno sicuramente ricordati linguagi quali: *Java* (1995), *C#* (2000), *Python* (1991).

## 1.4.2 Principali paradigmi di programmazione

Nei prossimi paragrafi verrano mostrati alcuni dei principali *paradigmi di programmazione*.

## 1.4.2.1 Programmazione imperativa

Questa tipologia di programmazione si basa sulla visione del programma come una sequenza di istruzioni da eseguire che vengono impartite all’elaboratore.  
Questo paradigma è, solitamente, il primo ad essere imparato da chi si approcia per la prima volta al mondo della programmazione vista la sua semplicità concettuale e l’ottima capacità di plasmare la *forma mentis* necessaria alla programmazione.  
*C*, *FORTRAN*, *Pascal*, sono solo alcuni degli esempi di linguaggi di programmazione che utilizzano questo paradigma.

## 1.4.2.2 Programmazione orientata agli oggetti

Questo paradigma di programmazione è concettualmente più complesso in quanto introduce i cosiddetti *oggetti software*.  
In un programma di questo tipo, gli *oggetti* interagiscono fra di loro scambiandosi messaggi che modificano il loro stato interno.  
Essi, inoltre, sono collegati fra loro mediante gerarchie di ereditarietà.  
Alcuni esempi di linguaggi correlati a questo paradigma sono: *Java, C#, Small Talk*.

## 1.4.2.3 Programmazione funzionale

In questo paradigma il programma è visto come   
una collezione di *funzioni matematiche* ognuna avente un proprio *dominio* ed un proprio *codominio*.   
Fondamentali per questa tipologia di programmazione sono concetti come la *composizione di funzioni* e la *ricorsione*.  
Alcuni esempi di linguaggi che utilizzano questo paradigma sono: *Lisp*, *Haskell*, *Scheme*.

1.4.2.4 Programmazione logica

Questa tipologia di programmazione si basa sulla descrizione della struttura logica del problema che si vuole affrontare, piuttosto che su come risolverlo.  
Questo è in forte contrapposizione rispetto ai sopracitati paradigmi.  
L’esempio principale di linguaggio di programmazione che utilizza questo paradigma è *Prolog*.

## 1.4.2.5 Programmazione visuale

Questo paradigma permette di scrivere programmi utilizzando, al posto del codice canonico, degli elementi grafici, quali simboli, disegni, ecc.  
La *programmazione visuale* ben si sposa bene sia con l’attività di approcio all’informatica per i più piccoli, utilizzando linguaggi come *Scratch, sia* per lo svolgimento di compiti complessi come, ad esempio, l’attività di simulazione di sistemi fisici, attraverso *Simulink*, come mostrato in Figura 1.2.

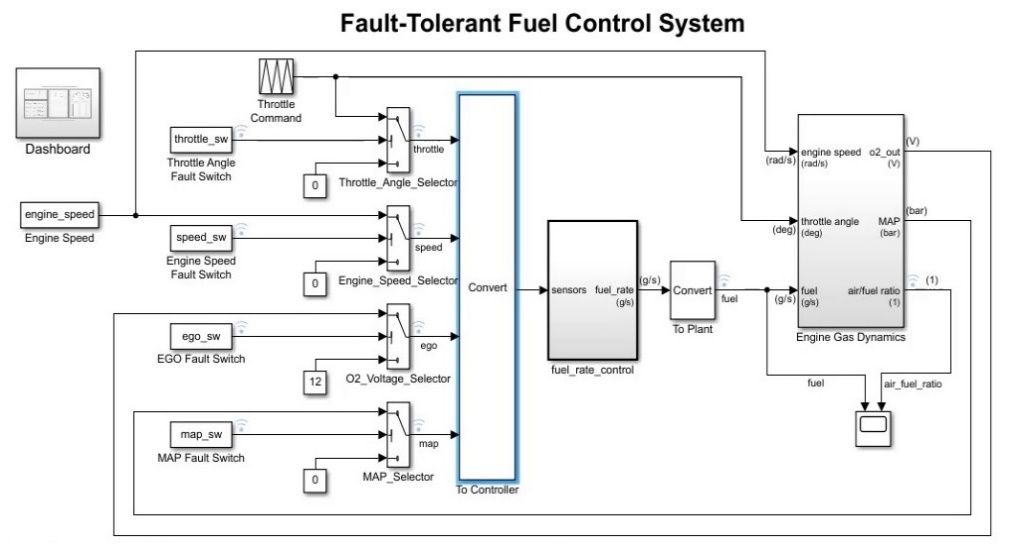


Figura 1.2: *Simulink*©MATLAB

# Capitolo 2 Specifica e implementazione di un linguaggio di programmazione

## 2.1 Specifica

La *specifica* di un linguaggio di programmazione è la descrizione delle sue caratteristiche principali, quali il *lessico*, la *sintassi* e la *semantica*.

Essa deve utilizzare una notazione rigorosa per evitare ambiguità, ma non deve eccedere né nella formalità né nell’informalità.  
Infatti, linguaggi come l’*ALGOL* furono rifiutati dalla *community* proprio per la loro descrizione eccessivamente formale.  
D’altro canto, un linguaggio non deve essere specificato in modo eccessivamente informale in quanto questo porta alla proliferazione di *dialetti*, come accadde al *Pascal*.

Il successo di un linguaggio è quindi strettamente legato alla sua *specifica* che deve necessariamente essere adatta a diversi destinatari, proprio per questa ragione si parla spesso di *specifica polimorfa*.  
Principalmente la *specifica* deve soddisfare tre categorie di destinatari: i *revisori*, gli *implementatori* e gli *utenti finali* del linguaggio.

I *revisori* sono coloro che, come per la pubblicazione di un qualunque articolo scientifico, si occupano di controllare e di fare una prima valutazione della *specifica* del linguaggio.  
Per i *revisori* è fondamentale che la *specifica* del linguaggio sia chiara ed elegante.

Gli *implementatori* si occupano invece dell’implementazione del linguaggio attraverso la scrittura di *traduttori*, *compilatori*, *macchine virtuali*, ecc.  
  
  
Per questa categoria risulta necessario che la *specifica* sia chiara nel sancire il significato di tutti i costrutti del linguaggio da implementare.

L’ultima categoria di destinatari è quella degli *utenti finali*, i quali sono i veri e propri fruitori del linguaggio di programmazione.  
Per essi è importante che il linguaggio sia specificato in modo semplice attraverso un *Reference Manual* contente esempi di utilizzo dei vari *costrutti*, oltre che essere di facile utilizzo e comprensione.

La *specifica* di un *linguaggio formale* ricalca, per certi aspetti, quella di un *linguaggio naturale*.  
In entrambi i casi bisogna definire il *lessico*, la *sintassi* e la *semantica* del linguaggio.  
Nei prossimi paragrafi si analizzerano in dettaglio questi tre processi fondamentali della *specifica*.

## 2.1.1 Lessico

Il *lessico* è l’insieme di tutte le parole che compongono un determinanto linguaggio, queste parole vengono dette *stringhe lessicali* e sono composte da caratteri appartenenti ad un determinato *alfabeto*.

Per specificare il *lessico* di un linguaggio è necessario introdurre la nozione di *simbolo.*Un *simbolo* è un’astrazione di una classe di *stringhe lessicali*, dove le singole *stringhe* sono dette *istanze* del *simbolo*.

La relazione fra un *simbolo* e le sue *istanze* è data da un *pattern*, cioè una regola che descrive come le *istanze* di un certo *simbolo* debbano essere costituite, come si vede in Figura 2.1.

Immagine che contiene tavolo

Descrizione generata automaticamente

Figura 2.: *Simbolo*, *istanze* e *pattern*©Gianfranco Lamperti

I *pattern*, nell’operazione di *specifica*, vengono definiti attraverso una serie di regole formali che danno vita alle cosiddette *espressioni regolari*.

## 2.1.1.1 Espressioni regolari

Le *espressioni regolari* sono un potente formalismo utlizzato per definire in modo rigoroso un *pattern*.

Esse si basano su una notazione simile a quelle delle espressioni aritmetiche (utilizzo di parentesi, operatori, proprietà algebriche, ecc.), con la sostanziale differenza che, al posto di ritornare un risultato numerico, esse ritornano un insieme di *stringhe*.  
 I principali operatori utilizzati nelle *espressioni regolari* sono:

* L’operatore di *concatenazione*, utilizzato per unire due o più caratteri di un *alfabeto*
* L’operatore di *opzionalità* “**|**”, utilizzato per l’operazione di scelta fra due o più caratteri
* L’operatore di *iterazione* “**\***”, utilizzato per ripetere un *carattere* zero o più volte

Altrettanto importante è il *carattere speciale* “**ε**” che indica il *carattere vuoto*.

Esempio: scrivere l’*espressione regolare* che definisce un numero binario: (0|1)(0|1)\*

## 2.1.1.2 Espressioni regolari estese

È possibile estendere le *espressioni regolari* introducendo nuovi operatori:

* L’operatore di *iterazione* “**+**”, utilizzato per ripetere una o più volte un carattere.
* L’operatore *qualsiasi carattere* “**.**”, utlizzato per indicare un qualunque carattere dell’*alfabeto*
* L’operatore *qualsiasi carattere con esclusione di caratteri* “**~**”, utilizzato per indicare un qualunque carattere dell’*alfabeto* all’infuori di alcuni caratteri specificati dopo il “**~**”
* L’operatore *range* “**[ ]**”, utlizzato per indicare tutti i caratteri in un determinato range
* L’operatore di *opzionalità* “**?**”, utilizzato per indicare che una sottoespressione è opzionale

Esempio: scrivere l’*espressione regolare* che definisce i commenti *Java-like non vuoti*   
 //(~\n)+\n

## 2.1.1.3 Definizioni regolari

Una *definzione regolare* è un’ulteriore estensione del concetto di *espressione regolare* che permette di assegnare un nome mnemonico ad un’*espressione regolare*.

Esempio: **lettera** → [A-Za-z]

## 2.1.1.4 Esempio

Di seguito viene riportato un esempio di come, attraverso le *definizioni* regolari, sia stato possibile specificare il *pattern* di definizione di una costante intera in *DLK*.

**cifra** → [0-9]  
**nozero** → [1-9]  
**intconst** → ((+|-)? **nozero cifra**\*) | 0

## 2.1.2 Sintassi