

Indice

1 Prefazione	1
1.1 Introduzione	1
1.2 Il progetto in breve	2
1.3 Digital Natives	2
1.4 Introduzione ai linguaggi di programmazione	2
1.4.1 Cenni storici sui linguaggi di programmazione	2
1.4.2 Principali paradigmi di programmazione	4
1.4.2.1 Programmazione imperativa	4
1.4.2.2 Programmazione orientata agli oggetti	4
1.4.2.3 Programmazione funzionale	5
1.4.2.4 Programmazione logica	5
1.4.2.5 Programmazione visuale	5
2 Specifica e implementazione di un linguaggio di programmazione	7
2.1 Specifica	7
2.1.1 Lessico	8
2.1.1.1 Espressioni regolari	9
2.1.1.2 Espressioni regolari estese	10
2.1.1.3 Definizioni regolari	10
2.1.1.4 Esempio	11
2.1.2 Sintassi	11
2.1.2.1 BNF	13
2.1.2.2 EBNF	14

2.1.2.3 Esempio	15
2.1.2.4 Diagrammi sintattici	15
2.1.3 Semantica.....	16
2.1.3.1 Semantica operativaale	16
2.1.3.2 Semantica denotazionale.....	17
2.2 Implementazione.....	18
2.2.1 Analisi lessicale	20
2.2.2 Analisi sintattica	20
2.2.2.1 Parsing top-down	22
2.2.3 Analisi semantica.....	23
2.2.4 Run-time	24
3 Linguaggi di programmazione per ragazzi	25
3.1 Introduzione	25
3.2 Logo.....	25
3.3 Scratch	27
4 DLK.....	29
4.1 Introduzione	29
4.2 Hello, world!	29
4.3 Struttura generale di un programma	30
4.4 Commenti.....	30
4.5 Variabili	31
4.5.1 Dichiarazioni.....	31
4.5.2 Assegnamento.....	33
4.5.3 Coercizione	34
4.6 Espressioni aritmetiche	34

4.6.1 Operatori aritmetici	35
4.6.1.1 Precedenza degli operatori aritmetici.....	36
4.6.1.2 Associatività degli operatori aritmetici	37
4.6.1.3 Parentesi nelle espressioni aritmetiche	38
4.6.1.4 Radice	38
4.7 Operazione di incremento e decremento di una variabile.....	39
4.8 Espressioni logiche	39
4.8.1 Operatori logici	40
4.8.2 Operatori relazionali	41
4.8.3 Operatori di uguaglianza.....	43
4.8.4 Precedenza degli operatori	43
4.8.5 Parentesi nelle espressioni logiche	44
4.9 L'istruzione se.....	45
4.9.1 La clausola altrimenti.....	46
4.9.2 Istruzioni se in cascata	47
4.10 L'istruzione ripeti	49
4.10.1 Cicli annidati.....	50
4.10.2 L'istruzione stop	50
4.11 L'istruzione scrivi	52
4.12 L'istruzione inserisci.....	53
4.13 L'interprete DLK	54
4.13.1 Come avviare l'interprete DLK	54
4.13.2 Come utilizzare l'interprete	55
4.13.3 Gestione degli errori	56
4.13.4 Avvertimenti	57

4.14 Repository	58
4.15 Esempio di programma scritto in DLK.....	58
5 Conclusione.....	60
5.1 Sviluppi futuri	60
Appendice A	61
A.1 Grammatica BNF del linguaggio DLK.....	61
A.2 Grammatica EBNF del linguaggio DLK	62
A.3 Definizioni regolari relative al lessico del linguaggio DLK.....	63
Bibliografia	64

Capitolo 1

Prefazione

1.1 Introduzione

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 questa tecnologia sia fondamentale per una realtà moderna e dinamica.

L'informatica 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.

In questa tesi si mostrerà un nuovo linguaggio di programmazione rivolto ai giovani, quale metodo moderno per avvicinare i ragazzi al mondo dell'informatica, trasformandoli da meri utilizzatori del "prodotto finito" a veri e propri *sviluppatori*.

1.2 Il progetto in breve

In questo lavoro verrà presentato un nuovo linguaggio di programmazione denominato *DLK* (*Didactical Language for Kids*).

Questo linguaggio si offre come un invito alla programmazione di alto livello rivolto ad un pubblico giovane, grazie all'utilizzo di costrutti semplificati, rispetto ai principali linguaggi di programmazione, e di *keyword* 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* (in italiano, *nativi digitali*) 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 espressione 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 di un elaboratore.

1.4.1 Cenni storici sui linguaggi di programmazione

I linguaggi di programmazione 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 poter essere eseguito, come mostrato in Figura 1.1.

Il *compilatore Assembly* viene spesso chiamato *Assembler* (*Assemblatore*).

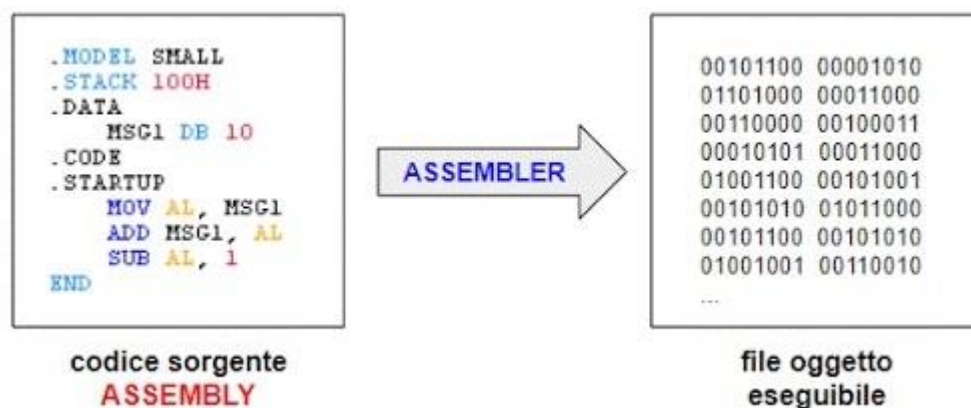


Figura 1.1: Traduzione in *linguaggio macchina* da *Assembly*
©andreaminini.com

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, introdusse il *paradigma di programmazione logico*.

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

1.4.2 Principali paradigmi di programmazione

Nei prossimi paragrafi verranno 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 insegnato a chi si avvicina 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#*, *Smalltalk*.

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 sia con l'attività di approccio 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.

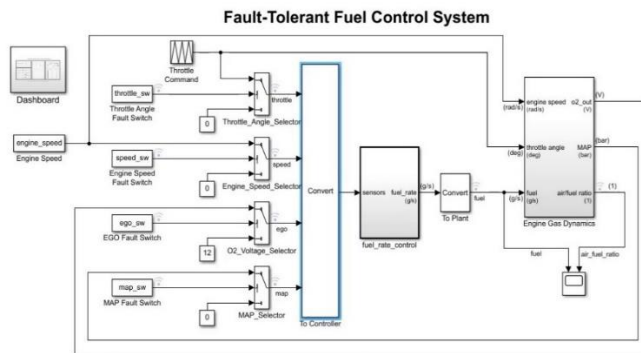


Figura 1.2: *Simulink*
©MATLAB

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 in rapporto allo stato dell'arte delle tecniche di *specifica* comunemente utilizzate.

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 adeguata a diverse tipologie di 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* contenente 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 analizzeranno in dettaglio questi tre processi fondamentali della *specifica*.

2.1.1 Lessico

Il *lessico* è l'insieme di tutte le parole che compongono un determinato 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.

<i>Simbolo</i>	<i>Istanze</i>	<i>Pattern</i>
while	while	while
begin	begin	begin
relop	< <= > >= != = ==	{ <, <=, >, >=, !=, =, == }
id	partenza tempo m24 X2	lettera seguita da lettere e/o cifre
num	3 25 3.5 4.37E12	parte intera seguita opzionalmente da parte decimale e/o parte esponenziale
strconst	"Hello world!"	sequenza di caratteri racchiusa tra "

Figura 2.1: *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 utilizzato per definire in modo rigoroso un *pattern*.

Esse si basano su una notazione simile a quella delle espressioni aritmetiche (utilizzo di parentesi, operatori, proprietà algebriche, ecc.), con la sostanziale differenza che, al posto di rappresentare un risultato numerico, esse rappresentano un insieme di *stringhe*.

I principali operatori utilizzati nelle *espressioni regolari* sono:

- L'operatore di *concatenazione*, utilizzato per unire due o più *espressioni regolari*.
- L'operatore di *opzionalità* “[|]”, utilizzato per l'operazione di scelta fra due o più *espressioni regolari*.
- L'operatore di *iterazione* “*”, utilizzato per ripetere un'*espressione regolare* 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'*espressione regolare*.
- L'operatore *qualsiasi carattere* ".", utilizzato 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 quelli inclusi nell'insieme rappresentato dal suo operando.
- L'operatore *range* "[]", utilizzato per indicare tutti i caratteri in un determinato range.
- L'operatore di *opzionalità* "?", utilizzato per indicare che una sotto espressione è opzionale.

Esempio: si scriva l'*espressione regolare* che definisce i commenti *Java-like* non vuoti.: $/(~\backslash n)^+\backslash n$

2.1.1.3 Definizioni regolari

Una *definizione regolare* è un'ulteriore estensione del concetto di *espressione regolare* che permette di definire una serie di associazioni tra nomi ed *espressioni regolari*.

Esempio:

lettera \rightarrow [A-Za-z]

cifra \rightarrow [0-9]

alfanumerico \rightarrow **lettera** | **cifra**

2.1.1.4 Esempio

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

cifra \rightarrow [0-9]

nozero \rightarrow [1-9]

intconst \rightarrow ((+|-)? **nozero cifra**^{*}) | 0

2.1.2 Sintassi

La *sintassi* di un linguaggio di programmazione è l'insieme di tutte le regole che permettono la descrizione della struttura delle *frasi* del linguaggio.

In altre parole, la *sintassi* indica come combinare le *stringhe lessicali* fra di loro in modo da formare *frasi* corrette (dal punto di vista sintattico).

A differenza della specifica della *sintassi* di un *linguaggio naturale*, quella di un *linguaggio artificiale* deve essere formata da un insieme di semplici regole limitate in numero.

Essa non tiene conto della specifica del *lessico*, mantenendo così nettamente separati *lessico* e *sintassi*.

Questo permette di migliorare la *portabilità* della *sintassi* stessa, in quanto se si decidesse

di modificare il *lessico* di un linguaggio, non si dovrebbe forzatamente agire sulla sua *sintassi*.

Inoltre, ciò permette di migliorare l'*efficienza* della *specifica* stessa utilizzando notazioni differenti e più specifiche per le regole sintattiche e per quelle lessicali.

Prima di entrare nel dettaglio di come avviene la *specifica sintattica* di un linguaggio, si vogliono introdurre due concetti fondamentali per la definizione formale di un linguaggio: il *riconoscimento* e la *generazione*.

Il *riconoscimento* è il meccanismo che permette di capire se una determinata *frase* appartiene o meno ad un linguaggio, come si vede in Figura 2.2.

Esso, preso da solo, non è molto utile alla definizione di un linguaggio, in quanto il suo funzionamento si basa su tentativi di riconoscimento di *frasi* potenzialmente appartenenti al linguaggio.

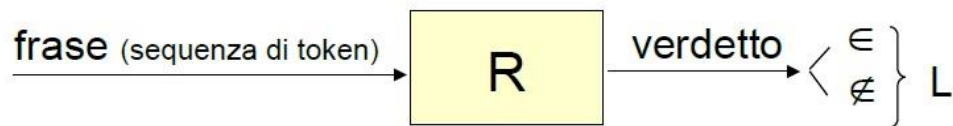


Figura 2.2: *Processo di riconoscimento*

©Gianfranco Lamperti

La *generazione*, invece, permette di creare una *frase* appartenente al linguaggio in modo del tutto casuale, come si vede in Figura 2.3.

Proprio in questa casualità risiede il principale problema di questo metodo, in quanto la frase generate sarà certamente corretta dal punto di vista *sintattico*, ma non da quello *semantico*.

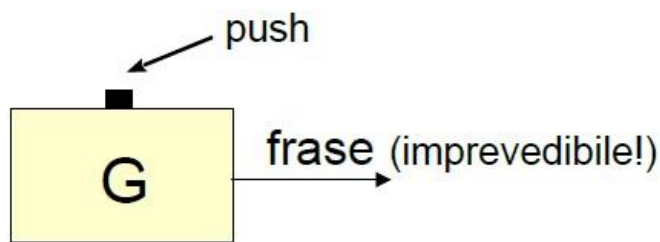


Figura 2.3: *Processo di generazione*

©Gianfranco Lamperti

I due concetti sopracitati sono all'apparenza sconnessi, ma grazie alle scoperte della *computer science*, sappiamo essere fortemente correlati.

Infatti il meccanismo di *riconoscimento* si basa su quello di *generazione*.

Esso cerca di generare la *frase* data in ingresso al meccanismo di *riconoscimento* e se il processo ha esito positivo, permette di affermare con certezza che la *frase* data appartiene al linguaggio.

Nei prossimi paragrafi si vogliono mostrare i principali formalismi utilizzati per la *specificazione sintattica*.

2.1.2.1 BNF

Negli anni '50 del Novecento, il linguista Avran Noam Chomsky e l'informatico John Warner Backus, già ideatore del *FORTRAN*, giunsero, in contesti del tutto separati, ad una conclusione del tutto equivalente al problema della *specificazione* della sintassi, la *BNF* (*Grammatica non contestuale*, per Chomsky).

Nel 1959 venne presentato il linguaggio di programmazione *ALGOL 58*, a cui lavorò intensamente Backus, la cui *sintassi* venne specificata, per la prima volta nella storia, attraverso la *BNF* (*Backus-Naur Form*).

Paradossalmente questo metodo non venne accettato prontamente dagli utenti dei linguaggi di programmazione, salvo poi diventare lo *standard* per la *specificazione sintattica*.

La *BNF* è, a tutti gli effetti, un linguaggio per descrivere un altro linguaggio, per questo viene spesso definita come un *metalinguaggio*.

La *BNF* di un linguaggio è un insieme di *produzioni* che mostrano, in modo astratto, le regole *grammaticali* del linguaggio e dei suoi costrutti.

Le *produzioni* sono costituite da un insieme di *simboli terminali*, ad esempio una *keyword* di un linguaggio di programmazione, e *nonterminali*, cioè richiami ad altre *produzioni*.

Una *produzione* può presentare diverse definizioni, per questa ragione è possibile utilizzare un operatore di *alternativa* “|” all’interno di una *produzione*.

È, inoltre, possibile utilizzare la *ricorsione* per definire delle liste o dei costrutti intrinsecamente ricorsivi come, ad esempio, le espressioni aritmetiche.

Esempio: scrivere la *produzione* relativa all’*if-statement* del *Pascal*

if-stat → **if** bool-expr **then** stat | **if** bool-expr **then** stat **else** stat

2.1.2.2 EBNF

È possibile estendere la *BNF* attraverso l’introduzione di nuovi operatori, dando così vita all’*EBNF*.

I principali operatori sono:

- L’operatore di *opzionalità* “[]”, attraverso il quale è possibile specificare se una parte di una *produzione* sia opzionale.
- L’operatore di *ripetizione* “{ }”, attraverso il quale è possibile ripetere una parte della *produzione*, evitando così di dover ricorrere alla *ricorsione* tipica delle *BNF*.
- L’operatore di *disgiunzione* “(|)”, utilizzato per la selezione di due o più possibilità all’interno di una *produzione*.

2.1.2.3 Esempio

In figura 2.4 viene riportato un esempio di confronto adoperato sulla stessa *produzione* scritta mediante la *BNF* e l'*EBNF*.

- BNF:

$$\begin{aligned} \text{expr} &\rightarrow \text{expr} + \text{term} \mid \text{expr} - \text{term} \mid \text{term} \\ \text{term} &\rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor} \end{aligned}$$


- EBNF:

$$\begin{aligned} \text{expr} &\rightarrow \text{term} \{ (+ \mid -) \text{term} \} \\ \text{term} &\rightarrow \text{factor} \{ (* \mid /) \text{factor} \} \end{aligned}$$

Figura 2.4: *BNF vs EBNF*

©Gianfranco Lamperti

2.1.2.4 Diagrammi sintattici

È possibile costruire *diagrammi sintattici* attraverso i quali visualizzare una *produzione*.

Essi utilizzano ovali per i *terminali* e rettangoli per i *nonterminali*, collegati da frecce, come si può vedere in Figura 2.5.

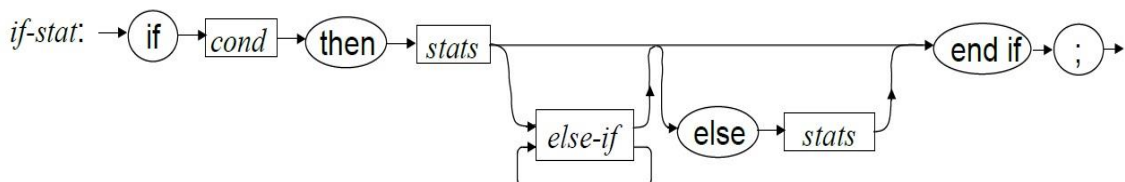


Figura 2.5: *Diagramma sintattico di un if-statement*

©Gianfranco Lamperti

2.1.3 Semantica

La *specifica* della *semantica* di un linguaggio di programmazione, a differenza delle precedenti, è molto meno *standardizzata*, in quanto non esiste un linguaggio di *specificazione* di riferimento.

Il compito di questa fase è quello di descrivere il significato dei *costrutti* del linguaggio di programmazione (*semantica dinamica*), attraverso l'utilizzo di un linguaggio e di costrutti di più *basso livello* rispetto al linguaggio da specificare.

I due principali formalismi per la *specificazione* della *semantica* sono: la *semantica operativa* e la *semantica denotazionale*, i quali verranno trattati nei prossimi paragrafi.

2.1.3.1 Semantica operativa

La *semantica operativa* è un formalismo utilizzato per descrivere la *semantica* di un linguaggio di programmazione dal punto di vista operativo.

Essa si basa sull'utilizzo di un linguaggio, o di uno pseudo-linguaggio, di più *basso livello* rispetto al linguaggio da descrivere, per mostrare *algoritmicamente* il significato di un *costrutto*, come si può vedere in Figura 2.6.

È bene ricordare che il linguaggio utilizzato per la *semantica operativa* è, normalmente, di tipo *imperativo*.

Istruzione C	Semantica operativa
<code>for(<i>expr</i>₁; <i>expr</i>₂; <i>expr</i>₃) <i>statements</i></code>	<code><i>expr</i>₁; loop: if <i>expr</i>₂ = 0 goto out <i>statements</i> <i>expr</i>₃; goto loop out:</code>

Figura 2.6: *Semantica operativa del ciclo for di C*

©Gianfranco Lamperti

2.1.3.2 Semantica denotazionale

La *semantica denotazionale* è un formalismo molto rigoroso, basato sulla teoria delle *funzioni ricorsive*, utilizzato per definire la *semantica* dei programmi e dei costrutti di un linguaggio di programmazione.

Per utilizzare questo formalismo è, innanzitutto, necessario individuare il *dominio matematico*, che non deve essere forzatamente numerico, delle *funzioni*.

Successivamente bisogna definire le *funzioni di mapping* che ad ogni istanza di un'*astrazione* (frammento di frase relativo a quell'*astrazione*) associano un elemento del *dominio matematico*.

Un altro concetto fondamentale per la *specificazione semantica* è lo *stato del programma*.

Questo è un insieme di coppie del tipo (i_n, v_n) , dove i_n è il nome di una *variabile* e v_n è il valore *corrente* della *variabile* i_n .

Per comodità, inoltre, si definisce una *funzione* $\mu(i_n, S)$ che dato il nome di una *variabile* i_n e lo *stato del programma* S , restituisce il valore v_n .

In Figura 2.7 viene mostrato un esempio di utilizzo della *semantica denotazionale* per descrivere l'operazione di assegnamento del *Pascal*.

$$\begin{aligned}
 M_a(x := E, s) = & \\
 & \text{if } M_e(E, s) = \text{errore} \text{ then} \\
 & \quad \text{errore} \\
 & \text{else} \\
 & \quad s' = \{(i_1, v_1'), (i_2, v_2'), \dots, (i_n, v_n')\}, \forall k \in [1 .. n] \left[v_k' = \begin{cases} \mu(i_k, s) & \text{if } i_k \neq x \\ M_e(E, s) & \text{otherwise} \end{cases} \right]
 \end{aligned}$$

confronto fra nomi!

Figura 2.7: *Semantica denotazionale dell'assegnamento del Pascal*

©Gianfranco Lamperti

2.2 Implementazione

L'*implementazione* di un linguaggio di programmazione è l'insieme di tutte quelle tecniche volte a “dare vita” al linguaggio stesso.

Infatti, mentre attraverso la *specifica* si definisce solamente come un linguaggio è strutturato dal punto di vista teorico, mediante la fase di *implementazione* è possibile rendere il linguaggio utilizzabile attraverso un *elaboratore*.

Solitamente, il processo di *implementazione* può seguire due tipologie di approccio differenti.

La prima tipologia di approccio riguarda la creazione di un *compilatore*, cioè di un *software* che riceve in *input* un programma scritto nel linguaggio da *implementare* (*linguaggio sorgente*) e ne restituisce una traduzione in un linguaggio già esistente (*linguaggio target*), come si vede in Figura 2.8.

Inoltre, è bene ricordare che nella maggior parte dei casi il *linguaggio target* è l'*Assembly*. Questa tipologia di implementazione è molto comune, ed è utilizzata da linguaggi come, ad esempio, il C.

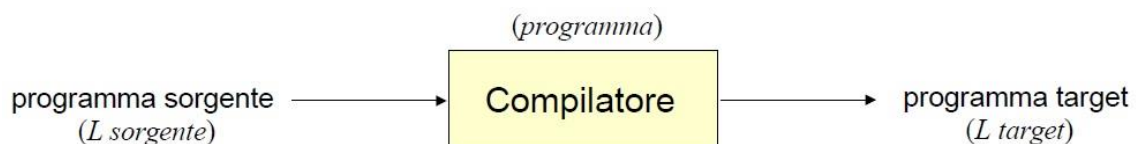


Figura 2.8: *Compilatore*

©Gianfranco Lamperti

La seconda tipologia di approccio, invece, si basa sulla creazione di un *interprete*, cioè un *software* in grado di eseguire direttamente il programma scritto nel linguaggio da *implementare*, senza doverlo tradurre in un altro linguaggio, come si può vedere in Figura 2.9.

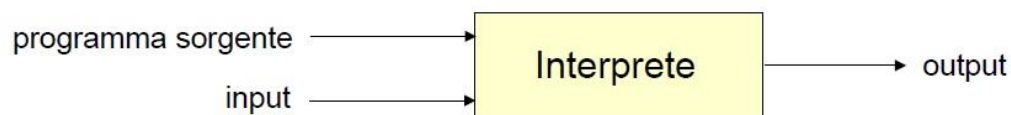


Figura 2.9: *Interprete*

©Gianfranco Lamperti

Vi è, inoltre, la possibilità di utilizzare soluzioni *ibride* basate sia su un *compilatore* che su un *interprete*, come si vede in Figura 2.10.

Queste soluzioni prevedono un *traduttore* che riceve in *input* il programma scritto nel linguaggio da *implementare* e lo trasforma in un linguaggio *intermedio*, come, ad esempio, il *Bytecode*.

Dopodiché il programma scritto nel linguaggio *intermedio* viene interpretato su di una *virtual machine*, come avviene, per esempio, con *Python*.

Inoltre, parte del *Bytecode* può essere, per migliorarne le prestazioni, compilato ad opera del *compilatore Just In Time*, come avviene nelle più recenti distribuzioni di *Java*.

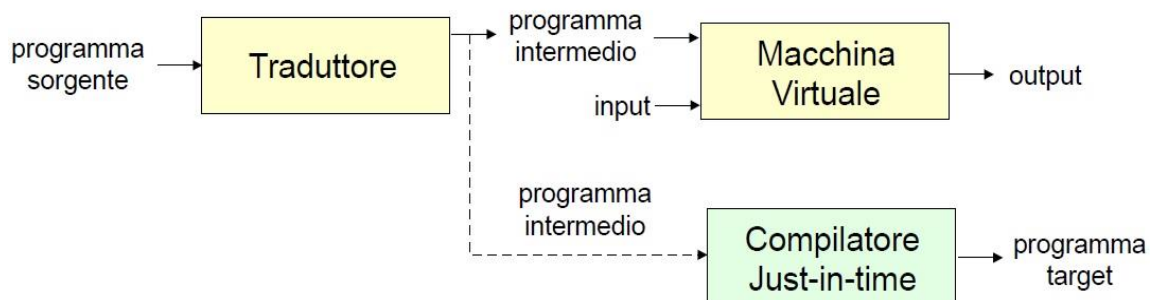


Figura 2.10: *Approccio ibrido*

©Gianfranco Lamperti

Nei prossimi paragrafi si tratterà delle tre fondamentali fasi comuni ai *compilatori* e agli *interpreti*: *analisi lessicale*, *analisi sintattica* e *analisi semantica*.

2.2.1 Analisi lessicale

L'*analisi lessicale* è il processo che prende in *input* il *codice sorgente* di un programma e restituisce la relativa sequenza di *token* (*simboli*).

Questo compito è svolto dal cosiddetto *analizzatore lessicale*, un programma che è in grado di riconoscere e catalogare i *token* contenuti nel *sorgente* da tradurre.

L'*analizzatore lessicale* più utilizzato è *Lex*.

Esempio: dato il seguente frammento di codice *C*, si mostra la lista di *token* creati dall'*analizzatore lessicale*.

```
int a = 3;
```

Token: {(INT), (ID, "a"), (EQ), (INTCONST, 3), (SEMICOLON)}

Si noti come un *token* è, generalmente, una coppia formata dal nome del *simbolo* e, opzionalmente, dal suo *valore*.

Il processo dell'*analisi lessicale* costituisce il primo passo fondamentale per la traduzione del *codice sorgente*, infatti fornisce al passaggio successivo, l'*analisi sintattica*, i *token* con i quali costruire l'*albero sintattico*.

È, inoltre, molto importante l'operazione di *controllo* del *lessico* che viene svolta durante l'*analisi lessicale*, attraverso la quale è possibile riconoscere eventuali elementi estranei al linguaggio.

Infine, in questa fase, vengono eliminati i *commenti* al codice, essendo ininfluenti sull'esecuzione del programma e, eventualmente, anche le spaziature.

2.2.2 Analisi sintattica

L'*analisi sintattica* è il processo attraverso il quale è possibile generare l'*albero sintattico* di un dato programma.

Questo compito è svolto dall'*analizzatore sintattico* o *parser*, un *software* che riceve in *input* i *token* generati dall'*analizzatore lessicale* e, riconoscendo le *derivazioni* utilizzate per costruire una *frase*, genera e restituisce l'*albero sintattico*.

L'*analizzatore sintattico* più utilizzato è *YACC*.

Una *derivazione* può essere interpretata come la modalità attraverso la quale una *produzione* genera una sua vera e propria *istanza*.

Questa può avere due forme: *derivazione canonica destra* o *derivazione canonica sinistra* a seconda che il *non terminale* sostituito sia quello più a *destra* o a *sinistra*.

Esempio: data la seguente *produzione* e data la seguente *frase*, si scriva come essa viene *derivata*.

Produzione: $A \rightarrow A + A \mid A - A$

Frase: **a - b + c**

Processo di derivazione: $A \Rightarrow A - A \Rightarrow \mathbf{a} - A \Rightarrow \mathbf{a} - A + A \Rightarrow \mathbf{a} - \mathbf{b} + A \Rightarrow \mathbf{a} - \mathbf{b} + \mathbf{c}$

L'*albero sintattico* è una rappresentazione grafica di una *derivazione* indipendentemente dall'ordine scelto nelle sostituzioni.

In Figura 2.11 è possibile vedere l'*albero sintattico* relativo all'esempio di *derivazione* precedente, infatti, leggendo le *foglie* da sinistra verso destra si trova la *frase* derivata.

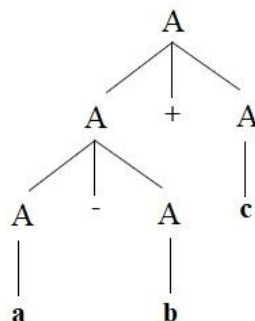


Figura 2.11: *Albero sintattico*

Durante questa fase, l'*analizzatore sintattico* è in grado di scovare gli errori di *sintassi* commessi dal programmatore.

Vi sono due principali tipologie di *analisi sintattica*: l'*analisi top-down* e quella *bottom-up*.

Nel prossimo paragrafo si tratterà del *parsing top-down*.

2.2.2.1 Parsing top-down

Il *parsing top-down* è una strategia di *analisi sintattica* che prevede la generazione dell'*albero sintattico* partendo dall'alto (*radice*) e muovendosi verso il basso, riconoscendo man mano le istanze delle *produzioni*, attraverso *derivazioni sinistre*.

Uno dei meccanismi di *parsing top-down* più utilizzato è quello del *parsing a discesa ricorsiva*.

Questo metodo si basa sull'analisi dell'*input* attraverso procedure *ricorsive*, che vengono solitamente create per ogni *nonterminale*.

La scelta della *produzione* da sviluppare viene presa attraverso l'analisi del *simbolo di lookahead*, cioè del simbolo successivo rispetto a quello che sta venendo correntemente analizzato.

Infatti se questo corrisponde ad un *nonterminale* viene chiamata la procedura relativa ad esso, altrimenti, se è un *terminale*, si controlla che vi sia un *match* con il *token* aspettato, come si vede in Figura 2.12.

stat → if *expr* then *stat* [else *stat*]

```
procedure stat()  
  match(IF); expr(); match(THEN); stat();  
  if lookahead = ELSE then  
    match(ELSE); stat()  
  endif  
end;
```

Figura 2.12: *Parsing a discesa ricorsiva del costrutto condizionale if*

©Gianfranco Lamperti

Esistono, inoltre, altri tipi di *parsing top-down* come, ad esempio, quelli della famiglia $LL(K)$, i quali si muovono da sinistra a destra (*Left to right scanning*), utilizzando *derivazioni sinistre* (*Leftmost derivation*) e scegliendo le *produzioni* da seguire in base a K , un numero naturale, *simboli di lookahead*.

2.2.3 Analisi semantica

L'*analisi semantica* è il processo attraverso il quale viene assicurata la correttezza *semantica* delle istruzioni utilizzate nel programma *sorgente*.

Questo processo è *statico* in quanto avviene durante la fase di *compilazione* del programma e quindi non ha potere sugli errori *run-time*.

Tipicamente le principali operazioni svolte in questa fase sono: la costruzione di una *symbol table* e il *type checking*.

La costruzione di una *symbol table* consta nella creazione di una *struttura dati* che mantiene informazioni relative agli *identificatori* utilizzati nel programma, ad esempio: *variabili, costanti, funzioni*, ecc.

Essi sono chiamati *simboli*.

Alcune delle principali informazioni sui *simboli* contenute nella *symbol table* sono: il *nome*, il *tipo*, il *valore* e lo *scope*, come si vede in Figura 2.13.

```

// Declare an external function
extern double bar(double x);

// Define a public function
double foo(int count)
{
    double sum = 0.0;

    // Sum all the values bar(1) to bar(count)
    for (int i = 1; i <= count; i++)
        sum += bar((double) i);
    return sum;
}

```

Symbol name	Type	Scope
bar	function, double	extern
x	double	function parameter
foo	function, double	global
count	int	function parameter
sum	double	block local
i	int	for-loop statement

Figura 2.13: Frammento di codice C con la relativa *symbol table*

©Wikipedia

Il *type checking* è l'insieme di tutte quelle operazioni che permettono di controllare se il *tipo* di un valore assegnato ad una *variabile* è coerente con il *tipo* dichiarato della variabile stessa.

Inoltre, questa operazione può essere applicata anche all'interno di *espressioni* e *costrutti*, ad esempio per controllare che in una somma tutti gli addendi siano numeri reali o in un *if-statement* per controllare che l'espressione *condizionale* restituisca un tipo *booleano*.

Attraverso il *type checking* e il mantenimento della *symbol table* è possibile individuare gli *errori semantici*, come, ad esempio, il tentativo di assegnare un valore ad una *variabile* non dichiarata in precedenza.

2.2.4 Run-time

La fase *run-time* indica l'intervallo temporale in cui un programma viene eseguito su un *elaboratore*.

A seconda che il linguaggio sia *compilato* o *interpretato*, questa fase può essere *indirettamente* o *direttamente* influenzata.

Il *compilatore*, infatti, non è in grado di controllare *direttamente* questa fase in quanto opera *staticamente* generando codice.

L'*interprete*, invece, può controllare *direttamente* questa fase in quanto mantiene all'interno delle sue *strutture dati* l'*ambiente di esecuzione* del programma.

Proprio per questa ragione, un *interprete* è in grado di individuare errori anche a *run-time*. Ad esempio, se l'utente inserisce un valore che porta ad una divisione per zero, l'*interprete* è in grado di segnalare questo errore, mentre un *compilatore* non sarebbe nella condizione di poterlo individuare (a meno che il divisore sia espresso dalla costante zero).

Capitolo 3

Linguaggi di programmazione per ragazzi

3.1 Introduzione

Negli anni '60, con l'avvento dei primi linguaggi di programmazione ad *alto livello*, alcune università, fra cui l'*MIT*, iniziarono a studiare apposite soluzioni al problema dell'insegnamento della programmazione ai più giovani.

Nacquero così i primi linguaggi di programmazione a scopo *didattico*.

Questi linguaggi si prefiggevano la volontà di facilitare lo sviluppo del cosiddetto *computational thinking* nei ragazzi, spesso utilizzando, per raggiungere questo scopo, *paradigmi visuali* piuttosto che *codice testuale*.

Nei prossimi paragrafi verranno introdotti due esempi di linguaggi *didattici*: *Logo* e *Scratch*.

3.2 Logo

Logo fu il primo linguaggio di programmazione realizzato con l'unico intento di avvicinare bambini e ragazzi al mondo della programmazione.

La sua ideazione risale al 1967 ad opera di Seymour Papert, Cynthia Solomon e Wallace Feurzeig [6].

Negli anni '70 vennero condotti alcuni esperimenti nelle scuole Americane introducendo l'insegnamento di questo linguaggio.

Tuttavia, a causa della scarsità di *calcolatori* disponibili per gli studenti, non fu così immediata la sua diffusione, per la quale si dovette aspettare fino agli anni '80 con l'avvento dei *Personal Computer*.

In principio, *Logo* permetteva, attraverso una serie di comandi *user-friendly*, di far muovere un *robot* dalle sembianze di una tartaruga che, grazie ad un pennarello incastonato sotto la “pancia” della tartaruga, era in grado di disegnare figure geometriche su di un foglio posizionato sul pavimento [7].

Nel 1969 venne implementata la prima *interfaccia grafica* per *Logo*, sostanzialmente la *tartaruga* da un oggetto fisico divenne un *cursore* di forma triangolare visibile sul *monitor* (chiamato anch’esso *turtle*) che, attraverso gli stessi comandi del *robot*, era in grado di disegnare figure geometriche sullo schermo, come si vede in Figura 3.1.

Una fondamentale caratteristica di questo linguaggio era data dal fatto che le istruzioni per far muovere il *cursore* venissero date tenendo conto del punto di vista della tartaruga e non del punto di vista “esterno”, o “dall’alto”, come solitamente avviene nel modo tradizionale di disegnare al *PC*.

Infatti in *Logo* le istruzioni di movimento venivano date in modo molto simile a quello utilizzato per indicare la strada ad un automobilista, per esempio il comando *RIGHT 90* permetteva di far svoltare il *cursore* di novanta gradi a destra.

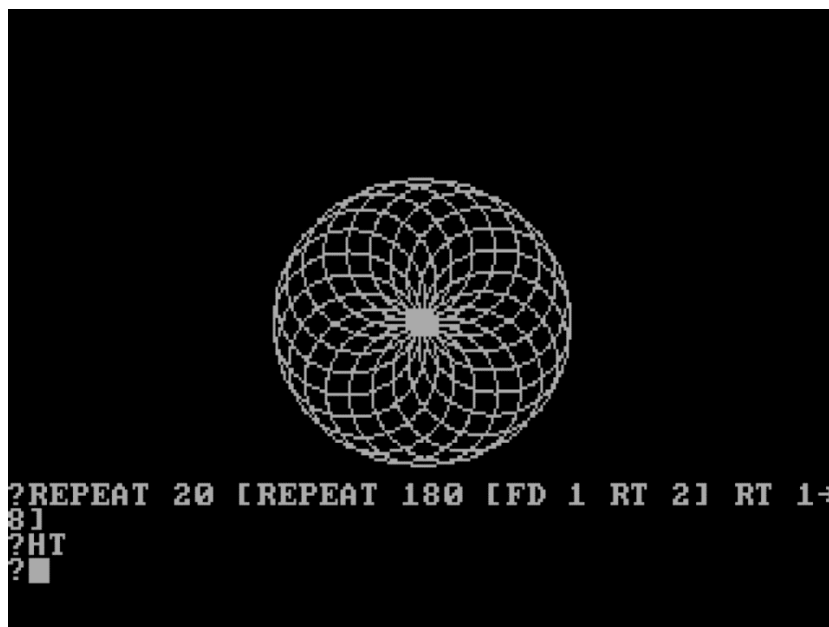


Figura 3.1: Codice sorgente Logo con il relativo output grafico

©Wikipedia.org

Negli anni successivi, a partire da *Logo* e grazie alla mancanza di un vero e proprio *standard* sono nate centinaia di *implementazioni* e di *dialetti*.

Alcune di esse hanno trasformato *Logo* in un linguaggio di programmazione “classico” allontanandosi così dall’idea di un linguaggio didattico.

Invece, altre *implementazioni* hanno incrementato il potere didattico di questo linguaggio introducendo il *paradigma* di programmazione *visuale* e migliorandone l’*interfaccia grafica*.

Una di queste è *Scratch*, di cui si parlerà nel prossimo paragrafo.

3.3 Scratch

Scratch nasce nel 2007, ad opera dei *media lab* del *MIT*, come un’estrema evoluzione del concetto di linguaggio di programmazione didattico introdotto da *Logo*.

Negli anni è stato tradotto in più di 70 lingue e vanta, ad oggi, più di 66 milioni di utenti iscritti [8].

Questo linguaggio utilizza un *paradigma* di programmazione *visuale*, basato su dei blocchi da combinare fra di loro per dare vita ad un programma, come si può vedere in Figura 3.2.

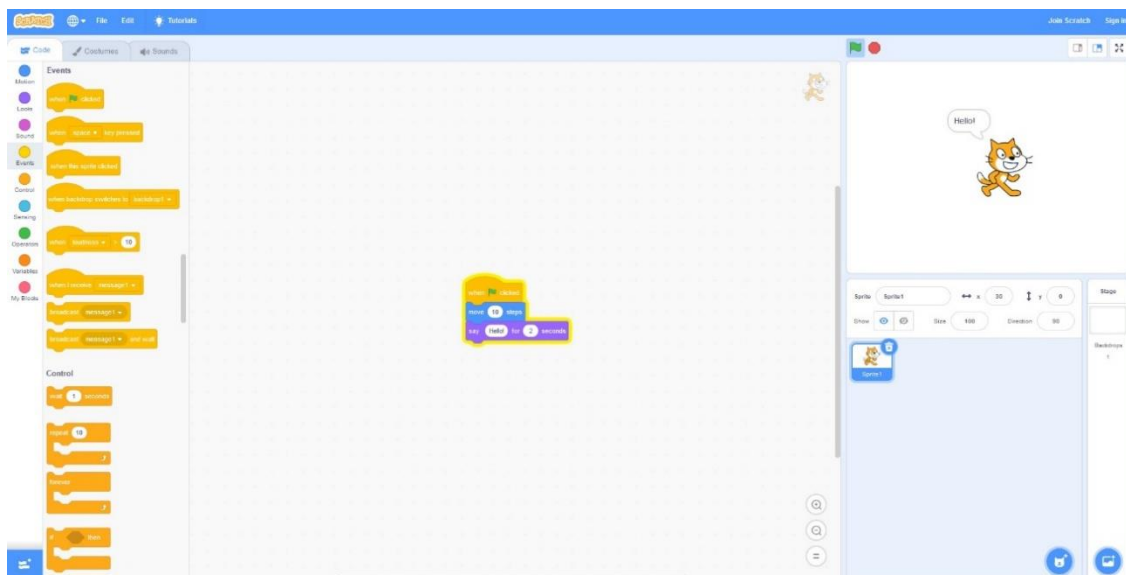


Figura 3.2: Codice sorgente Scratch con il relativo output

©Scratch

I programmi sono fondamentalmente delle animazioni di elementi grafici, di *default* un gatto, basate su *costrutti iterativi e condizionali*, oltre che su *variabili e procedure*. In questo linguaggio risulta relativamente facile progettare, ad esempio, un piccolo *videogioco*.

L'ambiente di sviluppo, oltre al classico utilizzo *offline*, permette di essere utilizzato anche completamente *online* all'indirizzo: <https://scratch.mit.edu>.

Va inoltre menzionato il particolare interesse verso questo progetto da parte di colossi dell'informatica quali *Google* e *AWS*.

Capitolo 4

DLK

4.1 Introduzione

Il linguaggio di programmazione *DLK*, acronimo di *Didactical Language for Kids*, nasce con l'intento di avvicinare i più giovani al mondo dello *sviluppo software*.

A differenza dei linguaggi di programmazione presentati nel precedente capitolo, *DLK* è stato ideato con la volontà di porsi ad un livello intermedio fra i linguaggi che utilizzano il *paradigma* di programmazione *visuale*, per essere il più possibile *user-friendly*, ed i linguaggi di programmazione *imperativi* di *alto livello*, come, ad esempio, il *C*.

Infatti, *DLK* permette di scrivere programmi utilizzando del *codice sorgente* relativamente facile, grazie all'utilizzo di *keyword* in italiano e *costrutti* semplificati.

Inoltre, *DLK* è un linguaggio *interpretato*, consentendo così la sua massima *portabilità*.

4.2 Hello, world!

Come da tradizione, il primo programma scritto attraverso *DLK* che verrà qui mostrato sarà quello che permette di *stampare* a video la frase “*Hello, world!*”.

inizio

 scrivi(“Hello, world!”);

fine.

Si noti che *DLK* non è *case sensitive* per quanto riguarda le *keyword*, quindi scrivere, ad esempio, **INIZIO** o **inizio** risulta essere la stessa cosa.

4.3 Struttura generale di un programma

Un programma scritto mediante il linguaggio *DLK* è generalmente costituito da due sezioni, ognuna delle quali ha un proprio ruolo.

La prima sezione è quella dedicata alla *dichiarazione* delle *variabili* utilizzate nel programma, in cui al nome di una *variabile* viene associato un unico *tipo* di dato.

Questa sezione è opzionale, in quanto se all'interno del programma non vengono utilizzate *variabili*, non vi è alcuna necessità di dichiararle.

La seconda sezione, invece, contiene le *istruzioni* che il programma deve svolgere, come *assegnamenti* e *costrutti*, ed è compresa fra le *keyword* “**inizio**” e “**fine.**”.

Questa sezione è obbligatoria, ma può esser lasciata vuota, senza scrivere alcuna *istruzione*.

In questo caso verrà comunicato un *avvertimento* da parte dell'*interprete*.

Nel linguaggio *DLK* ogni *istruzione* deve terminare con un “;”.

Qui di seguito viene mostrata la generica sintassi di un programma scritto in *DLK*.

sezione dedicata alla dichiarazione di variabili

inizio

istruzioni

fine.

4.4 Commenti

Quando si scrive del *codice*, sorge spontanea la necessità di aumentarne il più possibile la leggibilità e la comprensibilità, questo avviene sia grazie alle regole di *formattazione*, sia attraverso i *commenti*.

In *DLK* i *commenti* iniziano con “//”, seguiti da una qualunque sequenza di caratteri, e terminano quando si va a capo (*newline*).

È bene ricordare che i *commenti* vengono completamente ignorati da parte dell'*interprete*, quindi, ai fini dell'esecuzione del programma, sono ininfluenti.

Esempio: di commento in *DLK*.

```
// questo è un commento, verrà completamente ignorato dall'interprete
```

4.5 Variabili

I programmi, solitamente, prima di produrre un risultato in *output* necessitano di svolgere parecchie operazioni più o meno elementari.

Per questa ragione è necessario costruire dei “contenitori” in grado di memorizzare temporaneamente i dati durante l'esecuzione del programma.

Queste strutture di memorizzazione vengono chiamate *variabili*.

Innanzitutto, ad una *variabile* viene assegnato un nome, attraverso il quale sarà possibile accedere al dato in essa memorizzato (si noti che *DLK* è *case sensitive* per quanto riguarda il nome delle variabili).

In *DLK*, i nomi delle *variabili* sono stringhe *alfanumeriche* che iniziano con una lettera.

Per esempio, nomi corretti di *variabili* sono: a22, Riga1Colonna3, CONT.

4.5.1 Dichiarazioni

Una *variabile* per poter essere utilizzata all'interno di un programma, scritto mediante il linguaggio *DLK*, deve obbligatoriamente esser stata dichiarata in precedenza.

Come mostrato nel Paragrafo 4.3, esiste una sezione del codice appositamente dedicata alla dichiarazione delle *variabili*.

Qui di seguito viene riportata la sintassi dell'operazione di *dichiarazione* di una *variabile* in *DLK*.

tipo: nome della variabile;

Oltre ad un nome univoco, una *variabile* deve necessariamente avere un *tipo* che specifichi la tipologia di dati che potrà contenere.

In *DLK* i *tipi* utilizzabili sono quattro: *intero*, *decimale*, *stringa* e *boolean*.

- *Intero*: questo *tipo* è utilizzato per salvare numeri interi, come, per esempio, 22.
- *Decimale*: questo *tipo* è utilizzato per salvare numeri decimali, come 3.14. Si noti che la *virgola* in *DLK* è rappresentata da un “.”.
- *Stringa*: questo *tipo* è utilizzato per salvare stringhe, come, ad esempio “Hello, world!”.
- *Boolean*: questo *tipo* è utilizzato per salvare due soli valori, *vero* e *falso*.

È bene notare come, a differenza di linguaggi di programmazione come il *C*, *DLK* non abbia restrizioni legate al massimo e minimo numero rappresentabile, in quanto il suo interprete è scritto in *Python* e questo linguaggio non pone limiti di *range* ai numeri, se non quelli dettati dall’*hardware* della macchina sulla quale il programma è in esecuzione.

Esempio: dichiarare una variabile per ogni *tipo*.

intero: a;

decimale: b;

stringa: c;

boolean: d;

Vi è, inoltre, la possibilità di dichiarare una lista di *variabili* dello stesso tipo, separando i nomi delle *variabili* mediante una “,”.

Esempio: dichiarare tre variabili di tipo *intero* e una *boolean*.

intero: a, b, c;

boolean: d;

4.5.2 Assegnamento

L'operazione di *assegnamento* permette di conferire ad una *variabile* un certo valore coerente con il *tipo* dichiarato per quella *variabile*.

Per poter svolgere questa operazione è necessario che la *variabile* a cui si vuole assegnare un valore sia stata, in precedenza, *dichiarata*.

Qui di seguito viene riportata la sintassi dell'istruzione di *assegnamento* in *DLK*.

```
nome della variabile = costante;
```

Una volta assegnata una costante ad una *variabile*, se non viene modificata nel corso del programma, sarà possibile accedervi ogni qualvolta ci si riferisca al nome della *variabile*.

Esempio: assegnare valori alle seguenti *variabili*: a (*intero*), b (*decimale*), c (*stringa*), d (*boolean*).

```
a = 3;
```

```
b = 223.221;
```

```
c = "Ciao";
```

```
d = vero;
```

Oltre ad una costante è possibile assegnare ad una *variabile* il valore contenuto in un'altra *variabile*.

Inoltre, è possibile assegnare ad una *variabile*, di tipo *intero* o *decimale*, il risultato di un'*espressione aritmetica*.

Esempio: assegnare alla *variabile* a (*intero*) il valore della *variabile* b (*intero*) e alla *variabile* c (*intero*) il risultato dell'*espressione aritmetica* 2+2.

a = b;
c = 2 + 2;

4.5.3 Coercizione

La *coercizione*, o *conversione di tipo implicita*, è il meccanismo attraverso il quale l'*interprete* permette di assegnare *costanti* di tipo *intero* a variabili dichiarate *decimali* e viceversa.

Esempio: assegnare il valore 2 alla *variabile* di tipo *decimale* a.

a = 2;

Essendo 2 un numero *intero* e a una *variabile* di tipo *decimale*, l'*interprete* trasforma 2 in un numero *decimale*. Infatti, il valore che verrà salvato nella *variabile* a sarà 2.0.

È importante notare che questo meccanismo in *DLK* funziona solamente con i due tipi numerici, *intero* e *decimale*, e non con i tipi *stringa* e *boolean*.

4.6 Espressioni aritmetiche

DLK, come tutti i principali linguaggi di programmazione, offre la possibilità di utilizzare *espressioni aritmetiche* per calcolare valori numerici.

Esse sono utilizzate all'interno delle operazioni di *assegnamento* di un valore ad una *variabile*.

Esempio: di *espressione aritmetica*.

$(2 + j) - 11 * i / 2 - \text{radice}(k)$

Come si può notare dall'esempio, un'*espressione aritmetica* può esser costituita da *costanti, operatori aritmetici e parentesi*.

Nel prossimo paragrafo verranno presentati i vari tipi di *operatori aritmetici*.

4.6.1 Operatori aritmetici

In Tabella 4.1 vengono mostrati tutti i possibili *operatori aritmetici* del *DLK*.

Operatore	Tipo	Associatività
+, -, radice()	Unario	
+, -, *, /	Binario	Sinistra

Tabella 4.1: *Operatori aritmetici del DLK*

Gli *operatori unari* + e - vengono utilizzati per modificare il *segno* di un valore numerico, mentre l'operatore *radice* permette di calcolare la radice quadrata del valore, o dell'*espressione*, posti fra parentesi.

Esempio: di utilizzo degli *operatori unari*.

-radice(4)

Questa *espressione* risulta uguale a -2.0, in quanto la radice quadrata di 4 è uguale a 2.0 e l'operatore - ne cambia il segno.

Gli *operatori binari* + e - assumono il canonico significato di addizione e sottrazione, mentre * e / rappresentano, rispettivamente, i simboli di moltiplicazione e divisione.

Esempio: di utilizzo degli *operatori unari e binari* nella stessa *espressione*.

$$2 * 3 + -2$$

Questa *espressione* risulta uguale a 4

4.6.1.1 Precedenza degli operatori aritmetici

Quando un'*espressione aritmetica* contiene più *operatori* è necessario introdurre delle regole di *precedenza* per la valutazione di tali *operatori*.

Infatti, senza delle regole precise, risulta ambigua l'interpretazione di un'*espressione*, come si vede nell'esempio seguente.

Esempio: interpretare la seguente *espressione aritmetica*.

$$a + b * c$$

Questa *espressione* potrebbe essere interpretata come “sommo a e b e poi moltiplico per c” oppure “moltiplico b per c e poi sommo a”.

Entrambe le interpretazioni sono accettabili e proprio per evitare queste ambiguità risulta necessario sancire le regole di *precedenza*.

Nella Tabella 4.2 vengono presentati i diversi livelli di *precedenza* degli *operatori* del linguaggio *DLK*.

Operatore	Precedenza
+, -, radice() (unari)	Alta
*, /	Media

+, - (binari)	Bassa
---------------	-------

Tabella 4.2: *Ordine di precedenza degli operatori aritmetici del DLK*

Alla luce di quanto mostrato in Tabella 4.1, l'*espressione* dell'esempio precedente va interpretata come “moltiplico b per c e poi sommo a”.

4.6.1.2 Associatività degli operatori aritmetici

Oltre alle regole di *precedenza* degli operatori, è necessario sancire le regole di *associatività*.

Queste permettono di evitare ambiguità legate ad *operatori* con lo stesso livello di *precedenza*, come, ad esempio, * e /.

Esempio: interpretare la seguente *espressione aritmetica*.

$a * b / c$

In questo caso le regole di *precedenza* non danno informazioni utili su come interpretare questa *espressione*.

Potrebbe essere vista come “moltiplico a per b e poi divido per c” oppure come “divido b per c e poi moltiplico per a”.

In *DLK* tutti gli operatori *binari* sono *associativi a sinistra*, cioè, a parità di livello di *precedenza*, vengono raggruppati gli *operandi* più a sinistra.

Quindi l'esempio qui sopra andrebbe interpretato come “moltiplico a per b e poi divido per c”.

4.6.1.3 Parentesi nelle espressioni aritmetiche

In *DLK*, attraverso le parentesi tonde “()”, è possibile modificare il normale ordine di svolgimento delle *espressioni aritmetiche*.

Semplicemente, verrà svolta per prima l'*operazione aritmetica* inserita fra le parentesi, permettendo così di stravolgere le regole di *precedenza* e *associatività* degli *operatori*.

Esempio: di *espressione aritmetica* contenente le parentesi tonde.

$$-(a + b) * (c - d)$$

Questa espressione viene valutata in questo modo: “sommo a e b, cambio il segno del risultato e lo moltiplico per la differenza di c e d”.

Si noti che senza parentesi, ad esempio, b e c sarebbero state moltiplicate fra di loro prima di valutare la somma e la sottrazione.

4.6.1.4 Radice

Radice è un particolare *operatore aritmetico* che permette di calcolare la radice quadrata del *valore* posto fra parentesi.

Il *valore* deve essere numerico di tipo o *intero* o *decimale* e può essere il risultato di un'*espressione aritmetica*.

È bene notare che il risultato della *radice* è un numero di tipo *decimale*.

Esempio: di *espressione* contenente l'operatore *radice*.

$$\text{radice}((6 - 2) * 4)$$

Il risultato di questa *espressione* è dato dalla radice quadrata di 16, cioè 4.0 (si noti che è un numero di tipo *decimale*)

4.7 Operazione di incremento e decremento di una variabile

DLK offre la possibilità di incrementare, o di decrementare, di un'unità il valore contenuto all'interno di una *variabile* di tipo *intero* o *decimale*, attraverso due particolari *operatori*: ++ e --.

Questi operatori possono essere utilizzati all'interno del codice unicamente da soli, infatti non è possibile inserirli nelle *espressioni aritmetiche*.

Esempio: incrementare di un'unità la variabile *a* e decrementare di un'unità la variabile *b*.

```
a++;
```

```
b--;
```

Si noti come questo pezzo di codice sia equivalente alla scrittura estesa:

```
a = a + 1;
```

```
b = b - 1;
```

Per poter utilizzare questi operatori senza incorrere in errori, è necessario che la *variabile* a cui vengono applicati sia stata in precedenza *dichiarata* e gli sia stato assegnato un *valore* iniziale.

4.8 Espressioni logiche

In *DLK*, come nella maggior parte dei linguaggi di programmazione, vi è la necessità di valutare se una determinata *espressione* sia vera o falsa.

Per questa ragione *DLK* mette a disposizione le cosiddette *espressioni logiche*.

In *DLK* queste *espressioni* vengono utilizzate solamente nelle *istruzioni condizionali* denominate *se*, che verranno trattate in seguito.

È bene notare che il valore prodotto da un'*espressione logica* è sempre di tipo *boolean*, cioè o **vero** o **falso**.

Esempio: di *espressioni logiche*, con a, b, c variabili di tipo *intero*.

$a < b$

Questa *espressione* produrrà il valore *vero* se “ a è più piccolo di b ”, *falso* altrimenti.

$a < b$ e $b \neq c$

Questa *espressione* produrrà il valore *vero* se “ a è più piccolo di b e b è diverso da c ”, *falso* altrimenti. (Di seguito verranno spiegate le regole di *precedenza* degli *operatori*)

Nelle espressioni logiche entrano in gioco tre tipologie di operatori: gli *operatori logici*, gli *operatori relazionali* e gli *operatori di uguaglianza*.

4.8.1 Operatori logici

Gli *operatori logici* del *DLK* sono: **e** ed **o**, corrispondenti all’*and logico* e all’*or logico*, di cui, in Tabella 4.3 e 4.4, vengono mostrare le relative tabelle della verità.

A	B	A e B
vero	falso	falso
vero	vero	vero
falso	vero	falso
falso	falso	falso

Tabella 4.3: *Tabella della verità dell’operatore e*

A	B	A o B
vero	falso	vero

vero	vero	vero
falso	vero	vero
falso	falso	falso

Tabella 4.4: *Tabella della verità dell'operatore o*

Inoltre, gli *operatori logici* sono *associativi a sinistra* e l'*interprete DLK* li valuta in *cortocircuito*.

Esempio: di *espressione logica*, con $a = \text{vero}$, $b = \text{falso}$.

$a \text{ e } b \text{ o } a$

Questa *espressione* viene valutata, mediante le regole di *associatività*, in questo ordine: prima viene valutata “ $a \text{ e } b$ ”, il cui risultato è *falso* e poi viene valutata “*falso* o a ” che risulta *vero*.

Questi *operatori* andrebbero applicati a valori *booleani*, ma *DLK* permette anche di utilizzarli con valori *numerici* e *stringhe*.

I *numeri* vengono interpretati come *vero*, se sono diversi dallo zero, mentre, se sono uguali a zero, equivalgono al valore booleano *falso*.

Le *stringhe* vengono generalmente valutate come il valore booleano *vero*, solo nel caso di una *stringa vuota*, essa viene considerata come il valore *falso*.

4.8.2 Operatori relazionali

Gli *operatori relazionali* sono tutti quegli *operatori matematici* utilizzati per confrontare due valori.

Questi sono mostrati in Tabella 4.5 con il corrispondente significato.

Operatore	Significato
<	Minore di
<=	Minore o uguale a
>	Maggiore
>=	Maggiore o uguale a

Tabella 4.5: *Operatori relazione del DLK e loro significato*

Gli *operatori relazionali* nascono con l'intento di confrontare, fra loro, valori *numerici*, tuttavia in *DLK* è possibile applicarli anche a costanti di tipo *boolean* e *stringa*.

In caso vengano applicati a *stringhe*, il confronto viene fatto sulla lunghezza delle *stringhe*, mentre se vengono applicati a valori *booleani*, essi vengono visti come numeri, in particolare: *vero* viene interpretato come 1 e *falso* come 0.

Esempio: di utilizzo degli *operatori relazionali* con *stringhe*.

“ciao” > “ciao, come va?”

Questa *espressione* risulterà esser uguale al valore *falso*, in quanto la prima stringa è più corta della seconda.

Esempio: di utilizzo degli *operatori relazionali* con *variabili* contenenti valori di tipo *boolean*: a = *vero*, b = *falso*.

a <= b

Questa *espressione* risulterà esser uguale al valore *vero*, in quanto a verrà sostituita con 1 e b con 0.

4.8.3 Operatori di uguaglianza

Gli *operatori di uguaglianza* vengono utilizzati, come suggerisce il nome, per assicurarsi che due valori siano, o meno, identici.

Essi sono utilizzati, insieme a quelli *logici* e a quelli *relazionali*, per dar vita ad *espressioni logiche*.

DLK fornisce due *operatori* di questo tipo, come mostrato in Tabella 4.6.

Operatore	Significato
==	Uguale a
!=	Diverso da

Tabella 4.6: *Operatori di uguaglianza del DLK e loro significato*

Anche in questo caso, come con gli *operatori relazionali*, *DLK* permette di applicarli a valori di ogni tipo.

Esempio: di *espressioni* contenti *operatori di uguaglianza*.

“ciao” == “ciao”

Questa *espressione* risulterà esser *vera*, in quanto le due *stringhe* sono identiche.

vero != falso

Anche in questo caso l'*espressione* risulterà *vera*.

4.8.4 Precedenza degli operatori

Quando un'*espressione logica* contiene più *operatori* è necessario, come nel caso delle *espressioni aritmetiche*, introdurre delle regole di *precedenza* per la valutazione

degli *operatori*.

Infatti, senza delle regole precise, risulta ambigua l'interpretazione di un'*espressione*, come si vede nell'esempio seguente.

Esempio: interpretare la seguente *espressione logica*.

$a < b \text{ e } c$

L'interpretazione di questa *espressione* risulta ambigua, in quanto si potrebbe interpretare sia come “controllo che a sia minore di b e che c sia vero” che come “svolgo b e c e poi controllo che questo risultato sia maggiore di a ”.

Per questa ragione risulta fondamentale introdurre delle regole di *precedenza* degli *operatori*.

In Tabella 4.7 vengono riportati gli *operatori*: *logici*, *relazionali* e di *uguaglianza* con i relativi livelli di *precedenza*.

Operatore	Precedenza
$<, <=, >, >=, ==, !=$	Alta
e, o	Bassa

Tabella 4.7: *Ordine di precedenza degli operatori nelle espressioni logiche in DLK*

Alla luce di quanto mostrato in Tabella 4.7, l'*espressione* dell'esempio precedente va interpretata come “controllo che a sia minore di b e che c sia vero”.

4.8.5 Parentesi nelle espressioni logiche

In *DLK*, attraverso le parentesi tonde “()”, è possibile modificare il normale ordine di svolgimento delle *espressioni logiche*.

Come avviene per le *espressioni* aritmetiche, le *espressioni logiche* inserite fra parentesi

tonde verranno svolte per prime, permettendo così di stravolgere le regole di *precedenza* e *associatività* degli *operatori*: *logici*, *relazionali* e di *uguaglianza*.

Esempio: di *espressione logica* con parentesi.

a e (b o c)

Normalmente verrebbe valutata prima “a e b”, mentre, grazie alle parentesi, viene valutata per prima “b o c”.

4.9 L’istruzione se

L’istruzione *se* permette di scegliere, durante l’esecuzione di un programma, fra due alternative differenti sulla base del risultato di un’*espressione logica*.

Questa istruzione appartiene alla categoria delle istruzioni *condizionali*.

La sua sintassi è la seguente:

se (*espressione logica*) **vero fai:**

istruzioni

fine;

Come suggerisce la sintassi dell’istruzione stessa, se l’*espressione logica* produce come risultato *vero*, verranno svolte le *istruzioni*, altrimenti, se l’*espressione logica* produce come risultato *falso*, le *istruzioni* verranno ignorate.

Si noti che è possibile inserire nel corpo del *se* una o più *istruzioni* e, in caso non venga inserita alcuna *istruzione*, l’*interprete* lo segnalerà.

Esempio: di utilizzo dell’istruzione *se*, con corpo costituito da una sola *istruzione*.

se (a > b) vero fai:

a--;

fine;

c = 3;

Questo esempio va interpretato come: “se a è più grande di b, allora decremento di un’unità il valore di a”

Nel caso in cui a sia minore o uguale a b, invece, l’operazione di decremento non verrà svolta.

inoltre, si noti che l’istruzione di *assegnamento* del valore 3 a c viene svolta in ogni caso, essendo all’infuori del corpo dell’istruzione *se*.

Esempio: di utilizzo dell’istruzione *se*, con corpo costituito da più *istruzioni*.

se (a > b e c != 0) vero fai:

a--;

c = 0;

fine;

Questo esempio va interpretato come: “se a è maggiore di b e c è diverso da 0, allora decremento il valore di a di un’unità e assegno a c il valore 0”.

Se l’*espressione logica* dovesse risultare *falsa*, non verrebbe eseguita nessuna delle *istruzioni* contenute nel corpo dell’istruzione *se*.

4.9.1 La clausola altrimenti

DLK offre la possibilità di aggiungere opzionalmente all’istruzione *se* la clausola *altrimenti*.

Questa permette di svolgere delle *istruzioni* unicamente nel caso in cui l’*espressione logica* valutata nell’istruzione *se* risulti essere *falsa*.

La sintassi dell’istruzione *se* con l’aggiunta della clausola *altrimenti* è la seguente:

se (*espressione logica*) **vero fai:**

istruzioni

altrimenti:

istruzioni

fine;

Anche in questo caso, come con l'istruzione *se*, è possibile inserire nel corpo della clausola *altrimenti* una o più *istruzioni* e, se non viene inserita alcuna *istruzione*, l'*interprete* lo segnalerà.

Esempio: istruzione *se* con la clausola *altrimenti*.

se ($a > b$) vero fai:

`a--;`

altrimenti:

`a++;`

fine;

Questo esempio va interpretato come: “se a è più grande di b , allora incremento di un'unità il valore della variabile a , altrimenti, se a è minore o uguale di b , allora decremento di un'unità il valore di a ”.

4.9.2 Istruzioni se in cascata

Spesso, all'interno di un programma, risulta necessario controllare un gran numero di condizioni e fermarsi non appena una di queste risulti esser vera.

Per svolgere questo compito un'unica istruzione *se* con la clausola *altrimenti* risulta essere inadeguata, come si può vedere nel seguente esempio.

Esempio: si immagini di voler scrivere un programma in grado di riconoscere se un numero è positivo, negativo o nullo.

Attraverso un'unica istruzione *se* con la clausola *altrimenti* si potrebbero soltanto individuare due di queste condizioni e, al più, creare delle condizioni miste come “numero positivo o nullo”.

Per questa ragione in *DLK* è possibile creare istruzioni *se* in *cascata*, mediante la seguente *sintassi*.

se (*espressione logica*) **vero fai:**

istruzioni

altrimenti:

se (*espressione logica*) **vero fai:**

istruzioni

...

altrimenti:

istruzioni

fine;

(la keyword **fine**; va ripetuta una volta per ogni istruzione *se* utilizzata nella cascata)

Mediante questa tecnica è possibile, quindi, risolvere problemi come quello presentato nell'esempio all'inizio del paragrafo, come mostrato nel seguente esempio.

Esempio: di istruzioni *se* in *cascata*

num = 0;

se(num < 0) vero fai:

 // numero negativo

altrimenti:

 se(num > 0) vero fai:

```
// numero positivo  
  
altrimenti:  
  
    // numero nullo  
  
fine;  
  
fine;
```

Si noti che, visto l'utilizzo di due istruzioni *se*, il numero di “fine;” è due.

4.10 L'istruzione *ripeti*

Molto spesso, durante lo sviluppo di un programma, sorge il problema di dover *ripetere* più volte una o più istruzioni.

Si potrebbe pensare di risolvere questo tipo di problema riscrivendo le istruzioni tante volte quante le si vuole eseguire.

Questa soluzione, però, porterebbe ad aver molte linee di codice ripetute e ciò non gioverebbe né all'efficienza del programma né alla sua comprensibilità.

Per questa ragione, in tutti i principali linguaggi di programmazione sono presenti delle istruzioni di *ciclo* o *iterazione*.

In *DLK* vi è la possibilità di utilizzare come istruzione di *iterazione* il costrutto *ripeti*, di cui viene riportata qui sotto la sintassi.

ripeti *numero intero* **volte:**

istruzioni

fine;

Come suggerisce la sintassi stessa, le *istruzioni* contenute nel corpo del *ripeti* verranno eseguite per un numero di volte pari a quello indicato dal *numero intero*.

Le *istruzioni* contenute nel corpo del *ripeti*, come nel caso dell'istruzione *se*, possono essere una o più d'una e, in caso non venga inserita alcuna istruzione, l'*interprete* lo segnalerà al programmatore.

Esempio: si scriva un programma che tenga il conto del numero di *iterazioni* svolte dal costrutto *ripeti*.

```
i = 0;  
ripeti 10 volte:  
    i++;  
fine;
```

4.10.1 Cicli annidati

In *DLK* vi è la possibilità di richiamare un ciclo *ripeti* all'interno di un'altra istruzione *ripeti*.

Quando questa tecnica viene utilizzata, si parla di *cicli annidati*.

Esempio: di *cicli annidati*.

```
ripeti 2 volte:  
    scrivi("ciao");  
    ripeti 3 volte:  
        scrivi("\n");  
    fine;  
fine;
```

Questo esempio di codice stamperà a video due volte la parola “ciao” seguita, ogni volta, da 3 “a capo”.

4.10.2 L'istruzione stop

Alcune volte, durante l'esecuzione di un ciclo, vi è la necessità di abbandonarlo prima della sua normale terminazione.

Per questa ragione *DLK* mette a disposizione l'istruzione *stop*, di cui, di seguito, viene riportata la sintassi.

stop;

L'istruzione *stop* può essere utilizzata solamente all'interno del corpo dell'istruzione *ripeti*, altrimenti genera un errore.

È bene ricordare che l'utilizzo di istruzioni come *stop* è spesso sconsigliata in quanto rendono più complicato il *debug* dei programmi, ma, visto l'intento didattico del *DLK*, si è voluto comunque introdurre questa istruzione in quanto è di facile comprensione e utilizzo.

Esempio: di utilizzo dell'istruzione *stop*.

a = 0;

ripeti 10 volte:

 a++;

 se(a == 3) vero fai:

 stop;

 fine;

fine;

Si noti come, non appena la variabile *a* assume il valore 3, l'esecuzione del programma continui uscendo dal ciclo.

In questo caso, ci saranno solamente 3 iterazione del *ripeti*, prima che venga interrotto.

4.11 L'istruzione *scrivi*

L'istruzione *scrivi* permette di visualizzare sullo *standard output* un determinato valore appartenente ai *tipi* del *DLK*.

Di seguito viene riportata la sintassi di questa istruzione.

```
scrivi(valore);
```

Il *valore*, argomento dell'istruzione *scrivi*, può essere una qualsiasi *costante* numerica (*intero* o *decimale*) oppure di tipo *stringa*.

Inoltre, vi è la possibilità di passare come argomento all'istruzione *scrivi* il nome di una *variabile*, di cui verrà stampato il valore che essa contiene, indipendentemente dal suo *tipo*.

Esempio: di utilizzo dell'istruzione *scrivi*.

```
scrivi("ciao");
```

```
scrivi(1);
```

Questo segmento di codice stampa a video "ciao1".

Si noti, grazie al risultato dell'esempio, che l'istruzione *scrivi* non va "a capo" in automatico e, tanto meno, non "tabula".

Per questa ragione è necessario introdurre i cosiddetti *escape character*.

Gli *escape character* utilizzabili all'interno delle stringhe sono i seguenti:

- `"\n"`, utilizzato per andare "a capo".
- `"\t"`, utilizzato per "tabulare".

Esempio: di istruzione *scrivi* contenente *escape character*.


```
scrivi("ciao \ntutto bene?");
```

Il risultato di questa istruzione è il seguente:

ciao

tutto bene?

4.12 L'istruzione *inserisci*

L'istruzione *inserisci* permette di acquisire un valore che viene inserito da tastiera da parte dell'utente.

Di seguito viene riportata la sintassi di quest'istruzione.

```
inserisci(variabile);
```

Come suggerisce la sintassi, l'argomento dell'istruzione *inserisci* deve essere il nome di una *variabile*.

In questa *variabile* verrà salvato il valore inserito da tastiera dall'utente.

L'istruzione *inserisci*, non permette di inserire in una *variabile* dichiarata di un tipo, ad esempio *boolean*, un valore appartenente ad un altro tipo, come *stringa*.

Questo è possibile solamente con i due tipi numerici, *intero* e *decimale*, attraverso la coercizione.

Esempio: data la variabile di tipo *intero* *a*, si scriva l'istruzione che permette di inserire da tastiera il suo valore.

```
inserisci(a);
```

4.13 L'interprete DLK

L'*interprete DLK* è il *software* attraverso il quale è possibile eseguire i programmi scritti dall'utente nel linguaggio *DLK*.

Questo programma è stato scritto in *Python*, in quanto questo linguaggio ne permette la massima *portabilità*.

Per questo motivo è possibile utilizzarlo sia su sistemi operativi *Windows* che *Linux*.

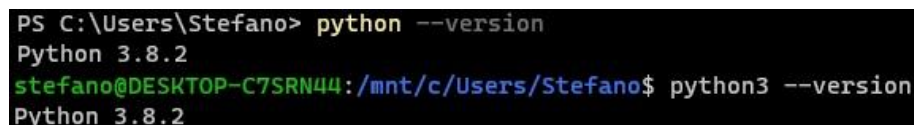
Nei prossimi paragrafi si mostrerà come utilizzare l'*interprete*.

4.13.1 Come avviare l'interprete DLK

Come prima cosa, su sistemi operativi *Windows*, bisogna scaricare ed installare *Python* (3.8 o più recenti) mediante il seguente link: <https://www.python.org/downloads>. Mentre sulle principali distribuzioni *Linux*, *Python* è già installato.

Per controllare che *Python* sia installato correttamente sul *computer*, si apra il *terminale* e si digiti il comando **python --version**, in *Windows*, mentre in *Linux* si digiti **python3 --version**.

Questo comando restituisce la versione installata di *Python*, come si vede in Figura 4.1.



```
PS C:\Users\Stefano> python --version
Python 3.8.2
stefano@DESKTOP-C7SRN44:/mnt/c/Users/Stefano$ python3 --version
Python 3.8.2
```

Figura 4.1: Comando per visualizzare la versione installata di *Python* in ambiente *Windows* (sopra) e in ambiente *Linux* (sotto)

Una volta installato *Python*, mediante il comando **cd** ci si rechi nella cartella contenente i file “DLK.py” e “Interpreter.py” contenenti il codice *Python* dell'*interprete DLK*.

Ora, mediante il comando **python DLK.py**, in *Windows*, oppure con il comando **python3 DLK.py**, in *Linux*, è possibile avviare l'*interprete* come si vede in Figura 4.2 e in Figura 4.3.

Una volta terminato il programma, l'*interprete* lo comunicherà e rimarrà in *stand-by* finché non verrà premuto il tasto *invio*, come si vede in Figura 4.5.

```
Inserire il nome del file contenente il programma da interpretare (ESC per uscire) --> prova.dlk
Questa è una prova!
Programma 'prova.dlk' terminato, premi invio per continuare|
```

Figura 4.5: Esempio di esecuzione di un semplice programma

Non appena verrà premuto *invio*, si ritornerà alla richiesta di inserimento del nome di un *file* da interpretare.

Se si vuole avviare un altro programma, si prosegue come illustrato in precedenza, altrimenti, per uscire dall'*interprete* si inserisce la *stringa* “ESC”, come mostrato in Figura 4.6.

```
Inserire il nome del file contenente il programma da interpretare (ESC per uscire) --> ESC
Arrivederci, a presto!
```

Figura 4.6: Terminazione dell'*interprete*

4.13.3 Gestione degli errori

Visto lo scopo didattico del linguaggio *DLK*, si è cercato di creare l'*interprete* nel modo più *user-friendly* possibile.

Particolare attenzione è stata posta sulla gestione degli errori che si manifestano durante l'interpretazione dei programmi scritti dall'utente.

Infatti, l'*interprete DLK* è in grado di riconoscere e comunicare nel modo più chiaro possibile le seguenti tipologie di errori:

- **Errori lessicali:** cioè tutti quegli errori legati all'utilizzo di caratteri non appartenenti all'alfabeto utilizzato da *DLK*.

- **Errori sintattici:** cioè tutti quegli errori derivanti dalla scorretta applicazione della grammatica del *DLK*, come, ad esempio, la dimenticanza del “.” alla fine del programma.
- **Errori run-time:** cioè tutti quegli errori che avvengono durante l'*esecuzione* del programma, come, ad esempio, il tentativo di accedere al valore di una *variabile* a cui non è stato *assegnato* alcun valore in precedenza.

Di seguito, in Figura 4.7, vengono riportati alcuni esempi di messaggi di errore forniti dall'*interprete*.

```
ERRORE DI SINTASSI:
Riga 5, colonna 1 --> ';' mancante

ERRORE DURANTE L'ESECUZIONE DEL PROGRAMMA:
Alla riga 9 --> Variabile 'z' non dichiarata

ERRORE DURANTE L'ESECUZIONE DEL PROGRAMMA:
Alla riga 7 --> L'argomento della 'radice' deve essere positivo

ERRORE DI SINTASSI:
Riga 7, colonna 11 --> Chiudere la parentesi ')'
```

Figura 4.7: Esempi di messaggi di errore da parte dell'*interprete*

4.13.4 Avvertimenti

Oltre agli errori, l'*interprete* è in grado di segnalare degli *avvertimenti* qualora ci si dimenticasse di scrivere delle *istruzioni* all'interno del *corpo* del programma o dei costrutti *se* e *ripeti*, come si può vedere in Figura 4.8.

```
ATTENZIONE:
--> Corpo del programma vuoto, non farà nulla!
```

Figura 4.8: Esempio di avvertimento

4.14 Repository

L'intero codice dell'*interprete DLK*, con i suoi futuri sviluppi, e alcuni esempi di programmi scritti mediante il linguaggio *DLK*, saranno resi disponibili al seguente link: <https://github.com/fraste97/DLK>.

4.15 Esempio di programma scritto in DLK

Qui di seguito verrà riportato un esempio di un programma scritto mediante l'utilizzo del linguaggio *DLK*.

Questo programma permette di calcolare l'area delle principali figure geometriche, quali: quadrati, rettangoli, triangoli e cerchi.

```
stringa: RICHIESTA, QUADRATO, BASE, ALTEZZA, RAGGIO, ERRORE, AREA,
AREANEGATIVA, SALUTI;
```

```
intero: scelta;
```

```
decimale: PIGRECO, lato, base, altezza, raggio, area;
```

```
boolean: ok;
```

```
inizio
```

```
PIGRECO = 3.1415;
```

```
RICHIESTA = "Inserire:\n1 per calcolare l'area di un quadrato\n2 per calcolare l'area di
un rettangolo\n3 per calcolare l'area di un triangolo\n4 per calcolare l'area di un
cerchio\n0 per uscire\n--> ";
```

```
QUADRATO = "\nInserire la lunghezza del lato --> ";
```

```
BASE = "\nInserire la lunghezza della base --> ";
```

```
ALTEZZA = "Inserire la lunghezza dell'altezza --> ";
```

```
RAGGIO = "\nInserire la lunghezza del raggio --> ";
```

```
ERRORE = "\nInserire un'opzione di scelta valida!\n\n";
```

```
AREA = "\nL'area è: ";
```

```
AREANEGATIVA = "\nAttenzione, l'area risulta essere negativa, inserire lunghezze
positive!\n\n";
```

```
SALUTI = "\nArrivederci, alla prossima!\n\n";
```

```
ripeti 1000 volte:
```

```
    ok = vero;
```

```
    scrivi(RICHIESTA);
```

```
    inserisci(scelta);
```

```
    se(scelta == 0) vero fai: //Esco dal programma
```

```
        stop;
```

```

    altrimenti:
        se(scelta == 1) vero fai: //Calcolo area quadrato
            scrivi(QUADRATO);
            inserisci(lato);
            area = lato*lato;
        altrimenti:
            se(scelta == 2) vero fai: //Calcolo area rettangolo
                scrivi(BASE);
                inserisci(base);
                scrivi(ALTEZZA);
                inserisci(altezza);
                area = base*altezza;
            altrimenti:
                se(scelta == 3) vero fai: //Calcolo area triangolo
                    scrivi(BASE);
                    inserisci(base);
                    scrivi(ALTEZZA);
                    inserisci(altezza);
                    area = base*altezza/2;
                altrimenti:
                    se(scelta == 4) vero fai: //Calcolo area cerchio
                        scrivi(RAGGIO);
                        inserisci(raggio);
                        area = raggio*raggio*PIGRECO;
                    altrimenti: //Non è stato inserito un valore corretto
                        scrivi(ERRORE);
                        ok = falso;
                fine;
            fine;
        fine;
    fine;
    se(ok) vero fai: //Controllo non ci siano stati errori
        se(area >= 0) vero fai:
            //Presentazione risultato
            scrivi(AREA);
            scrivi(area);
            scrivi("\n\n");
        altrimenti:
            scrivi(AREANEGATIVA);
        fine;
    fine;
fine;
scrivi(SALUTI);
fine.

```

Capitolo 5

Conclusione

5.1 Sviluppi futuri

Il progetto in oggetto a questa tesi ben si presta a potenziali futuri sviluppi, sia dal punto di vista dell'*interprete* che del linguaggio *DLK* stesso.

Innanzitutto, si potrebbe pensare di estendere il linguaggio di programmazione, introducendo nuove istruzioni e strutture dati.

Ad esempio si potrebbero ideare: un ciclo *while*, un'istruzione per calcolare le potenze di un numero oppure introdurre gli *array*.

Di conseguenza anche l'*interprete* andrebbe modificato per sopperire alle modifiche della *grammatica* del linguaggio.

Inoltre, l'*interprete* potrebbe essere reso ancor più *user-friendly* mediante lo sviluppo di un'*interfaccia grafica* e potenziando la sua capacità di segnalazione degli errori, introducendo suggerimenti da affiancare ai messaggi di errore.

Un altro potenziale progetto futuro, strettamente legato al linguaggio *DLK*, potrebbe esser dato dall'implementazione di un *ambiente di sviluppo integrato (IDE)* nel quale poter scrivere il proprio codice *DLK*, assistiti da suggerimenti e correzioni automatici. Questo *IDE* potrebbe essere sviluppato sia per esser eseguito su *PC* che sotto forma di *applicazione web*, rendendolo così accessibile ovunque.

Si potrebbe anche creare un sito *internet* da cui poter scaricare l'*interprete* e visionare l'intera documentazione del linguaggio *DLK*, affiancandolo con un canale *YouTube* in cui caricare *tutorial* sulla programmazione in *DLK*.

In fine, sarebbe interessante, nonché di grande aiuto, consultare un *pedagogo*, con il quale individuare quali potrebbero essere le migliori soluzioni future per rendere il linguaggio *DLK* ancora più adatto al modo di apprendere dei bambini e dei ragazzi.

Appendice A

A.1 Grammatica BNF del linguaggio DLK

$program \rightarrow decl-list \mid body$

$decl-list \rightarrow decl \ ; \ decl-list \mid \epsilon$

$decl \rightarrow type \ : \ id-list$

$type \rightarrow \text{intero} \mid \text{decimale} \mid \text{stringa} \mid \text{boolean}$

$id-list \rightarrow id \ , \ id-list \mid id$

$body \rightarrow \text{inizio} \ stat-list \ \text{fine} \ .$

$stat-list \rightarrow stat \ ; \ stat-list \mid \epsilon$

$stat \rightarrow assign-stat \mid se-stat \mid ripeti-stat \mid \text{stop} \mid scrivi-stat \mid inserisci-stat \mid inc-dec-stat$

$assign-stat \rightarrow id = rhs-assign-stat$

$rhs-assign-stat \rightarrow math-expr \mid \text{bool-const} \mid \text{string-const}$

$bool-const \rightarrow \text{vero} \mid \text{falso}$

$math-expr \rightarrow math-expr + math-term \mid math-expr - math-term \mid math-term$

$math-term \rightarrow math-term * math-factor \mid math-term / math-factor \mid math-factor$

$math-factor \rightarrow (math-expr) \mid radice-stat \mid \text{num-const}$

$radice-stat \rightarrow \text{radice} \ (\ math-expr)$

$num-const \rightarrow \text{intconst} \mid \text{realconst} \mid id$

$se-stat \rightarrow \text{se} \ (\ logical-expr \) \ \text{vero fai} \ : \ stat-list \ \text{altrimenti-stat} \ \text{fine}$

$altrimenti-stat \rightarrow \text{altrimenti} \ : \ stat-list \mid \epsilon$

$logical-expr \rightarrow logical-expr \ \text{e} \ rel-expr \mid logical-expr \ \text{o} \ rel-expr \mid rel-expr$

$rel-expr \rightarrow rel-term < rel-term \mid rel-term \leq rel-term \mid rel-term > rel-term \mid$

$rel-term \geq rel-term \mid rel-term == rel-term \mid rel-term != rel-term \mid rel-term$

$rel-term \rightarrow num-const \mid string-const \mid id \mid (\ logical-expr \)$

$ripeti-stat \rightarrow \text{ripeti} \ \text{intconst} \ \text{volte} \ : \ stat-list \ \text{fine}$

$scrivi-stat \rightarrow \text{scrivi} \ (\ scrivi-arg \)$

$scrivi-arg \rightarrow \text{strconst} \mid id \mid \text{num-const}$

$inserisci-stat \rightarrow \text{inserisci} \ (\ id \)$

$inc-dec-stat \rightarrow id++ \mid id--$

A.2 Grammatica EBNF del linguaggio DLK

$program \rightarrow decl-list \mid body$
 $decl-list \rightarrow \{decl \ ;\}$
 $decl \rightarrow type : id-list$
 $type \rightarrow \text{intero} \mid \text{decimale} \mid \text{stringa} \mid \text{boolean}$
 $id-list \rightarrow \text{id} \{ , id \}$
 $body \rightarrow \text{inizio} \ stat-list \ \text{fine} \ .$
 $stat-list \rightarrow \{ stat \ ;\}$
 $stat \rightarrow assign-stat \mid se-stat \mid ripeti-stat \mid \text{stop} \mid scrivi-stat \mid inserisci-stat \mid$
 $\quad inc-dec-stat$
 $assign-stat \rightarrow \text{id} = rhs-assign-stat$
 $rhs-assign-stat \rightarrow math-expr \mid \text{bool-const} \mid \text{string-const}$
 $bool-const \rightarrow \text{vero} \mid \text{falso}$
 $math-expr \rightarrow math-term \{ (+ \mid -) math-term \}$
 $math-term \rightarrow math-factor \{ (* \mid /) math-factor \}$
 $math-factor \rightarrow (math-expr) \mid radice-stat \mid \text{num-const}$
 $radice-stat \rightarrow \text{radice} (math-expr)$
 $num-const \rightarrow \text{intconst} \mid \text{realconst} \mid \text{id}$
 $se-stat \rightarrow \text{se} (logical-expr) \ \text{vero fai} : stat-list \ [altrimenti-stat] \ \text{fine}$
 $altrimenti-stat \rightarrow \text{altrimenti} : stat-list$
 $logical-expr \rightarrow rel-expr \{ (\text{e} \mid \text{o}) rel-expr \}$
 $rel-expr \rightarrow rel-term \ [(< \mid \leq \mid > \mid \geq \mid == \mid !=) rel-term]$
 $rel-term \rightarrow num-const \mid string-const \mid \text{id} \mid (logical-expr)$
 $ripeti-stat \rightarrow \text{ripeti intconst volte} : stat-list \ \text{fine}$
 $scrivi-stat \rightarrow \text{scrivi} (scrivi-arg)$
 $scrivi-arg \rightarrow \text{strconst} \mid \text{id} \mid \text{num-const}$
 $inserisci-stat \rightarrow \text{inserisci} (\text{id})$
 $inc-dec-stat \rightarrow \text{id}++ \mid \text{id}--$

A.3 Definizioni regolari relative al lessico del linguaggio DLK

- **id**

lettera \rightarrow [A-Za-z]

cifra \rightarrow [0-9]

alfanum \rightarrow lettera | cifra

id \rightarrow lettera alfanum*

- **intconst**

cifra \rightarrow [0-9]

nozero \rightarrow [1-9]

segno \rightarrow +|-

intconst \rightarrow (segno? nozero cifra*) | 0

- **realconst**

cifra \rightarrow [0-9]

nozero \rightarrow [1-9]

segno \rightarrow +|-

intconst \rightarrow segno? (nozero cifra*) | 0

realconst \rightarrow intconst ('.' (cifra)+)?

- **strconst**

strconst \rightarrow \" (~[\"'])*\"

- **commento**

commento \rightarrow //(~[\n])* \n

Bibliografia

- [1] Marc Prensky. Digital Natives, Digital Immigrants. On the Horizon, MCB University Press, Vol 9 No.5. 2001.
- [2] Paolo Ferri. Nativi digitali. Mondadori. 2011
- [3] Paolo Ferri. Nativi digitali puri e nativi digitali spuri. 2011.
- [4] Ada Lovelace. Nota G. 1953.
- [5] Raúl Rojas, Cüneyt Göktekin, Gerald Friedland, Mike Krüger. The First High-Level Programming Language and its Implementation. 2000.
- [6] Logo Foundation. What is Logo?. 2015. URL: https://el.media.mit.edu/logo-foundation/what_is_logo/history.html
- [7] Cyberneticzoo.com. 1969 - The Logo turtle - Seymour Papert et al. 2010. URL: <http://cyberneticzoo.com/cyberneticanimals/1969-the-logo-turtle-seymour-papert-marvin-minsky-et-al-american/>
- [8] MIT Media Lab. Scratch's statistics. URL: <https://scratch.mit.edu/statistics/>