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.
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, 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'utilizzo 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 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*).

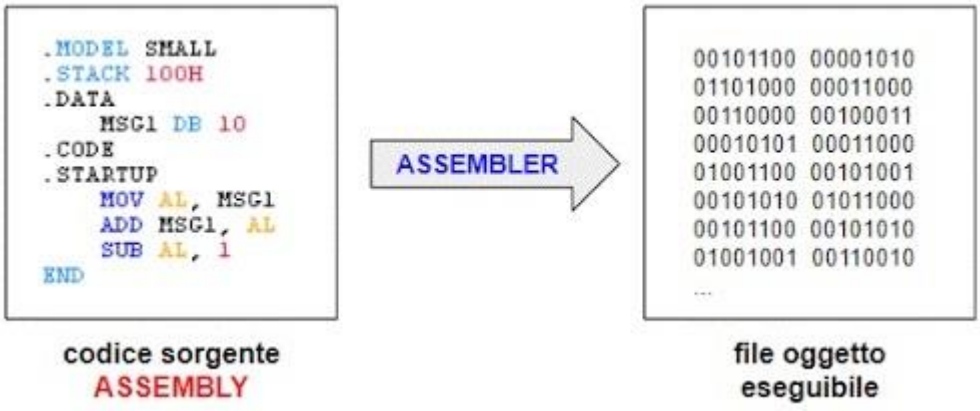

Figura 0.1.1: Traduzione in *linguaggio macchina* da *Assembly*
©andreaminini.com

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 approccia 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 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.

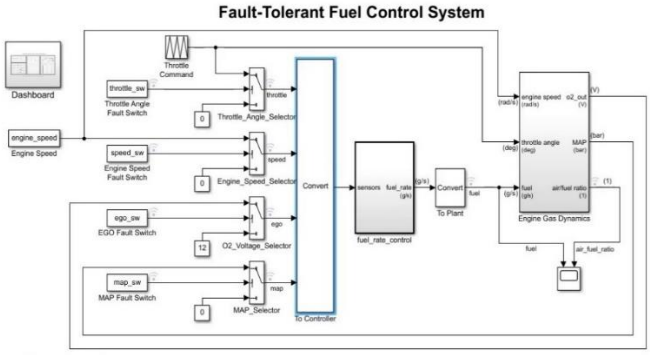

Figura 1.2: *Simulink*

©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.
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* 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 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.

Simbolo	*Istanze*	*Pattern*
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*

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 ritornare un risultato numerico, esse restituiscono 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* "**.**", 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 alcuni caratteri specificati dopo il "**~**".
- 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.: $//(\sim\backslash n)^+\backslash n$

2.1.1.3 Definizioni regolari

Una *definizione regolare* è un'ulteriore estensione del concetto di *espressione regolare* che permette di 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

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.

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.

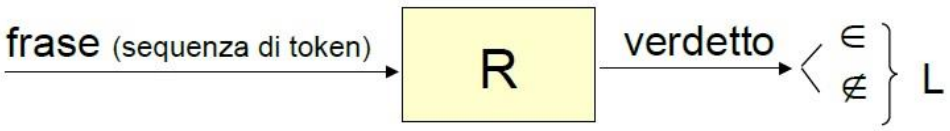

Figura 2.2: *Processo di riconoscimento*
©Gianfranco Lamperti

La *generazione*, invece, permette di creare una *frase* appartenete 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*.

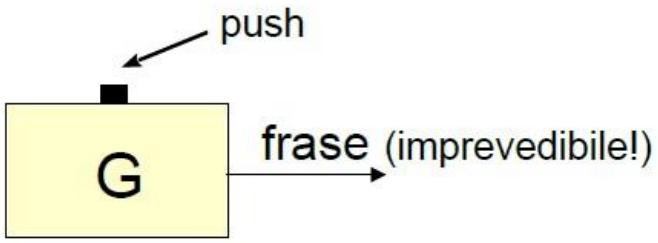

Figura 2.3: *Processo di generazione*
©Gianfranco Lamperti

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 *specifica lessicale*.

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 *specifica* 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 *specifica 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 *non terminali*, cioè richiami ad altri *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.

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*.

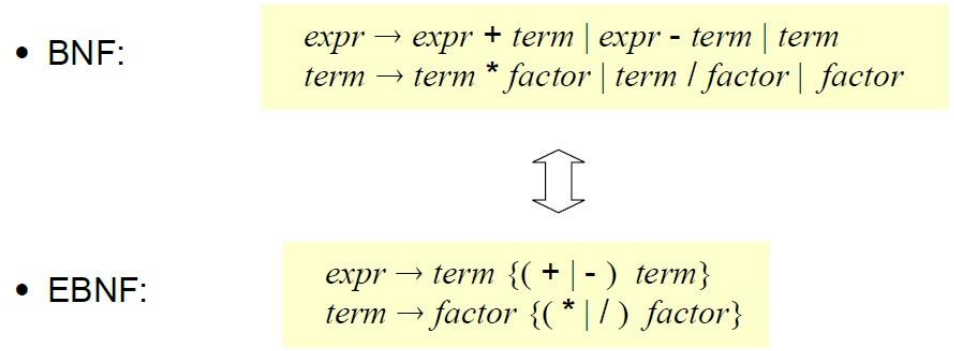

Figura 2.4: *BNF vs EBNF*

©Gianfranco Lamperti

2.1.2.4 Diagrammi sintattici

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

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

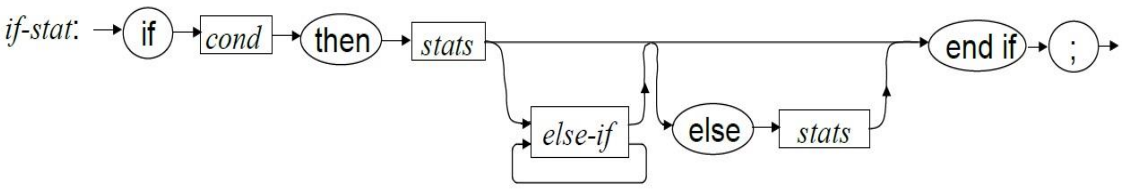

Figura 2.5: *Diagramma sintattico di un if-statement*

©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 *specifica* 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 *specifica* della *semantica* sono: la *semantica operazionale* e la *semantica denotazionale*, i quali verranno trattati nei prossimi paragrafi.

2.1.3.1 Semantica operazionale

La *semantica operazionale* 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 operazionale* è, normalmente, di tipo *imperativo*.

Istruzione C	*Semantica operazionale*

`for(`$expr_1$`;` $expr_2$`;` $expr_3$`)`
statements
	$expr_1$`;`
	`loop: if` $expr_2$ `= 0 goto out`
	statements
	$expr_3$`;`
	`goto loop`
	`out:`

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

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 *specifica 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* **μ(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*.

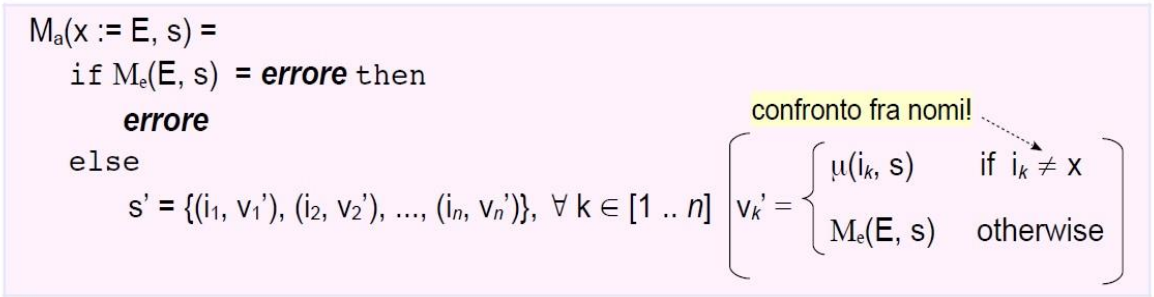

Figura 2.7: *Semantica denotazionale dell'assegnamento del Pascal*
©Gianfranco Lamperti

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 *calcolatore*.
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*.

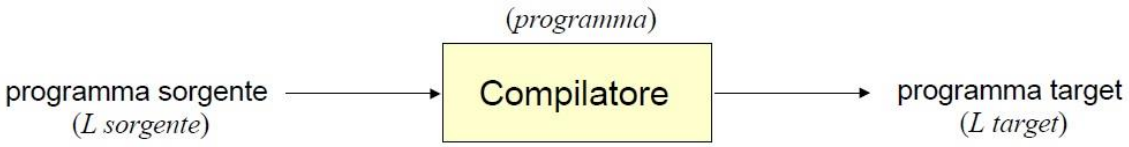

Figura 2.8: *Compilatore*
©Gianfranco Lamperti

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.

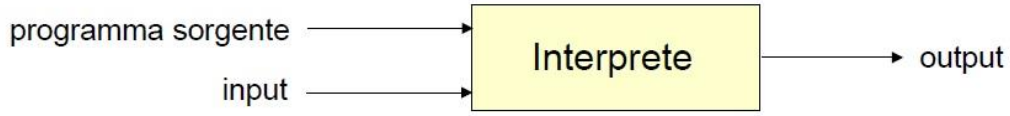

Figura 2.9: *Interprete*

©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*.

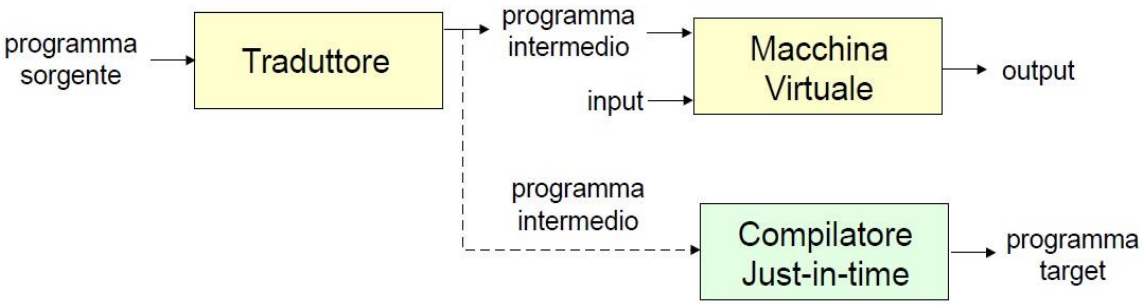

Figura 2.10: *Approccio ibrido*

©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, attraverso il principio del *maximal munch*, 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 → A + A | A − A
Frase: **a − b + c**
Processo di derivazione: A → A − A → **a** − A → **a** − A + A → **a − b** + A → **a − b + 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.

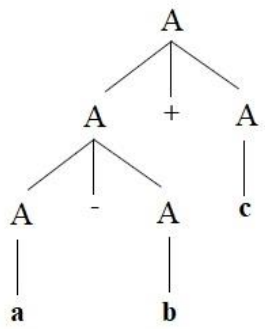

Figura 2.11: *Albero sintattico*

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 lo sviluppo 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 *non terminale*.
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 *non terminale* viene chiamata la procedure 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 ai *nomi mnemonici* utilizzati nel programma di: *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.

```c
// 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 il momento in cui un programma per *elaboratore* viene eseguito.
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 genera codice *staticamente*.

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.

Capitolo 3

Linguaggi di programmazione per ragazzi

3.1 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.

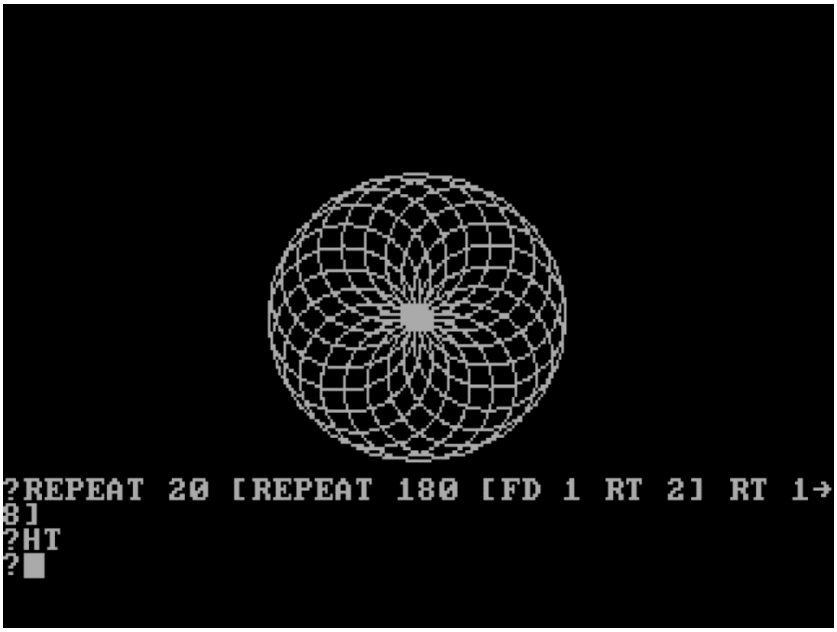

Figura 3.1: *Codice sorgente Logo con il relativo output grafico*
©Wikipedia.org

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.

3.2 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.
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*.

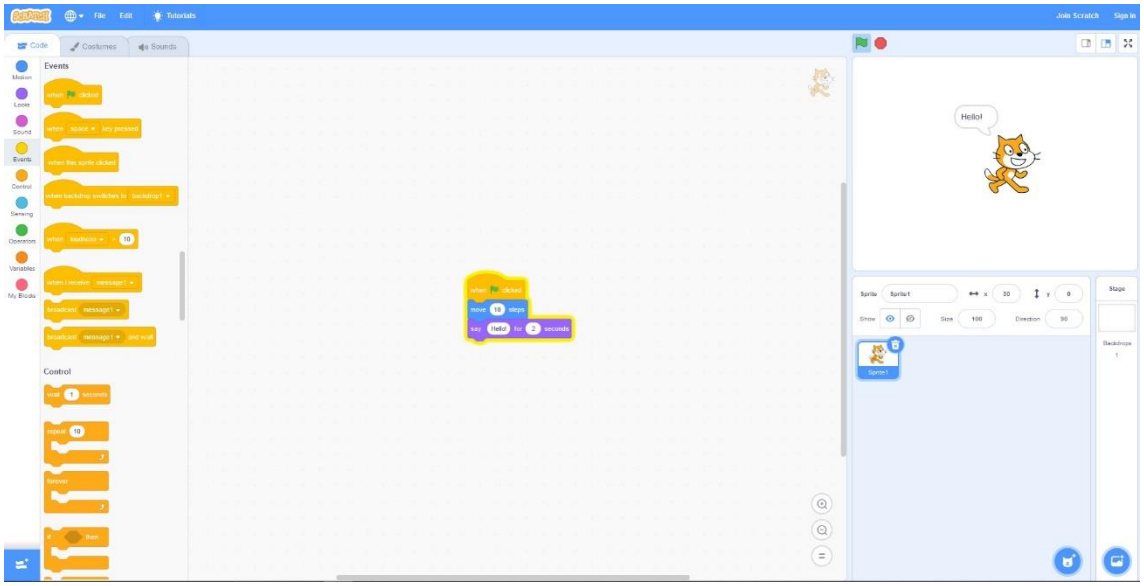

Figura 3.2: *Codice sorgente Scratch con il relativo output*
©Scratch

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